Higher-Order LCTRSs and Their Termination

No Author Given

No Institute Given

Abstract Logically constrained term rewriting systems (LCTRSs) are a formalism for program analysis with support for data types that are not (co)inductively defined. Only imperative programs have been considered through the lens of LCTRSs so far since LCTRSs were introduced as a first-order formalism. In this paper, we propose logically constrained simply-typed term rewriting systems (LCSTRSs), a higher-order generalization of LCTRSs, which suits the needs of representing and analyzing functional programs. We also study the termination problem of LCSTRSs and define a variant of the higher-order recursive path ordering (HORPO) for the newly proposed formalism.

Keywords: Higher-order term rewriting · Constraints · Recursive path ordering.

1 Introduction

It is hardly a surprising idea that term rewriting can serve as a vehicle for reasoning about programs. During the last decade or so, the term rewriting community has seen a line of work that translates real-world problems from program analysis into questions about term rewriting systems, which include for instance termination (see, e.g., [8,10,15,31]) and equivalence (see, e.g., [13,30,9]). Such applications take place across programming paradigms due to the versatile nature of term rewriting, and often materialize into automatable solutions.

Data types are a central building block of programs and must be properly handled in program analysis. While it is rarely a problem for term rewriting systems to represent (co)inductively defined data types, others such as integers and arrays traditionally require encoding; e.g., representing -3 as $\mathsf{neg}\ (\mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ 0)))$. This usually turns out to cause more obfuscation than clarification to the methods applied and the results obtained. An alternative is to incorporate primitive data types into the formalism, which contributes to the proliferation of subtly different formalisms that are generally incompatible with each other, and it is often difficult to reuse techniques developed in a distinctive setting.

Logically constrained term rewriting systems (LCTRSs) [24,12] emerged from this proliferation as a unifying formalism seeking to be general in both the selection of primitive data types (little is presumed) and the applicability of varied methods (many are extensible). LCTRSs thus allow us to benefit from the broad term rewriting arsenal in a wide range of scenarios for program analysis.

As a first-order formalism, LCTRSs only naturally accommodate imperative programs. This paper aims to generalize LCTRSs in a higher-order setting.

Motivation. One of the defining features of functional programs is their accommodation of higher-order functions, which are occasionally exploited to fine-tune the control flow of a program, as demonstrated in the style of an LCTRS below:

$$\mathsf{fact}\ n\ k \to k\ 1 \qquad [n \leq 0] \qquad \mathsf{comp}\ g\ f\ x \to g\ (f\ x)$$
$$\mathsf{fact}\ n\ k \to \mathsf{fact}\ (n-1)\ (\mathsf{comp}\ k\ ((*)\ n)) \qquad [n > 0]$$

where $n \leq 0$ and $n > 0$ are logical constraints, which the integer n must satisfy respectively when the corresponding rewrite rule is applied. These rewrite rules do not actually fit into an LCTRS because an explicit continuation k of type $\mathsf{int} \to \mathsf{int}$ is passed as an argument, whereas LCTRSs are a first-order formalism.

This small example, albeit artificial, shows some characteristics of functional programs as well as the limit of LCTRSs' expressivity. Nevertheless, we would like an automatable solution to the termination problem of functional programs in the same fashion as the first-order case [24,22], or even better, a way of automatically determining their complexity through rewriting methods.

Moreover, given two programs supposedly implementing the same function, a method that tells whether they are indeed equivalent is also desirable. Consider, for example, the more straightforward implementation of the factorial function:

$$\mathsf{fact}\ n \to 1 \quad [n \leq 0] \qquad \mathsf{fact}\ n \to n * \mathsf{fact}\ (n-1) \quad [n > 0]$$

A proof that the two are equivalent (up to an identity function passed as the continuation) may serve as a correctness proof of the former, less intuitive implementation. Such methods have been explored in a first-order setting [6,12].

Higher-order LCTRSs will broaden the horizons of both LCTRSs, and higher-order term rewriting systems. The eventual goal is to have a formalism that can be deployed to analyze both imperative and functional programs, so that through this formalism, the abundant techniques based on term rewriting may be applied to automatic program analysis. This paper is a step toward that goal.

Contributions. Our presentation begins with the perspective of this paper on higher-order term rewriting (without logical constraints) in Section 2. Then:

- We propose the formalism of *logically constrained simply-typed term rewriting systems* (LCSTRSs), a higher-order generalization of LCTRSs, in Section 3.
- We adapt the notions of a *reduction ordering* and *rule removal* to LCSTRSs, and provide (and prove correctness of) a definition of *constrained HORPO*—a variant of the higher-order recursive path ordering [20]—in Section 4. This includes changes to adapt HORPO to curried notation, and to handle theory symbols and constraints similar to RPO for LCTRSs [24]. While this version of HORPO is not the most powerful higher-order termination technique, it provides a simple yet self-contained solution, and serves to illustrate how existing term rewriting techniques may be extended to the new formalism.
- We have developed the foundation of a new (open-source) analysis tool for LCSTRSs, and provide an implementation of constrained HORPO, which is discussed in Section 5. This requires several new insights, especially with regards to the way the theory and constraints are handled.

2 Preliminaries

One of the first problems that a student of higher-order term rewriting faces is the absence of a standard formalism on which the literature agrees. This variety reflects the diverse interests and needs held by different authors.

In this section, we present *Simply-typed Term Rewriting Systems* [25] as the unconstrained basis of our formalism. This is one of the simplest higher-order formalisms, and closely resembles simple functional programs. We have chosen STRSs as a starting point because they are already very powerful, while avoiding many of the complications that may be interesting for equational reasoning but are not needed for program analysis, such as reduction modulo β.

Types and terms. Types rule out undesired terms. We consider *simple types*: given a non-empty set $\mathcal{S}$ of *sorts* (or *base types*), the set $\mathcal{T}$ of simple types over $\mathcal{S}$ is generated by the grammar $\mathcal{T} ::= \mathcal{S} \mid (\mathcal{T} \to \mathcal{T})$. Right-associativity is assigned to $\to$ so we can omit some parentheses. The *order* of a type A, denoted by $\mathrm{ord}(A)$, is defined as follows: $\mathrm{ord}(A) = 0$ for $A \in \mathcal{S}$ and $\mathrm{ord}(A \to B) = \max(\mathrm{ord}(A) + 1, \mathrm{ord}(B))$.

Given disjoint sets $\mathcal{F}$ and $\mathcal{V}$, whose elements we call *function symbols* and *variables*, respectively, the set $\mathfrak{T}$ of *pre-terms* over $\mathcal{F}$ and $\mathcal{V}$ is generated by the grammar $\mathfrak{T} ::= \mathcal{F} \mid \mathcal{V} \mid (\mathfrak{T}\ \mathfrak{T})$. Left-associativity is assigned to the juxtaposition operation, called *application*, so, e.g., $f\ x\ y$ should be read $(f\ x)\ y$.

We assume that every function symbol and variable is assigned a unique type. Typing applications works as expected: if pre-terms t_0 and t_1 have types $A \to B$ and A, respectively, $t_0\ t_1$ has type B. The set of *terms* over $\mathcal{F}$ and $\mathcal{V}$, denoted by $T(\mathcal{F}, \mathcal{V})$, is the subset of $\mathfrak{T}$ consisting of pre-terms with a type. We write $t : A$ if a term t has type A. The set of variables occurring in a term $t \in T(\mathcal{F}, \mathcal{V})$, denoted by $\mathrm{Var}(t)$, is defined as follows: $\mathrm{Var}(f) = \emptyset$ for $f \in \mathcal{F}$, $\mathrm{Var}(x) = \{ x \}$ for $x \in \mathcal{V}$ and $\mathrm{Var}(t_0\ t_1) = \mathrm{Var}(t_0) \cup \mathrm{Var}(t_1)$. A term t is called *ground* if $\mathrm{Var}(t) = \emptyset$. The set of ground terms over $\mathcal{F}$ is denoted by $T(\mathcal{F}, \emptyset)$.

Substitution and contexts. Variables occurring in a term can be seen as placeholders: variables may be replaced with terms which have the same type as the variable does. Type-preserving mappings from $\mathcal{V}$ to $T(\mathcal{F}, \mathcal{V})$ are called *substitutions*. Every substitution σ extends to a type-preserving mapping $\bar{\sigma}$ from $T(\mathcal{F}, \mathcal{V})$ to $T(\mathcal{F}, \mathcal{V})$. We write $t\sigma$ for $\bar{\sigma}(t)$ and define it as follows: $f\sigma = f$ for $f \in \mathcal{F}$, $x\sigma = \sigma(x)$ for $x \in \mathcal{V}$ and $(t_0\ t_1)\sigma = (t_0\sigma)\ (t_1\sigma)$.

Term formation gives rise to the concept of a context: a term containing a hole. Formally, let $\Box$ be a special terminal symbol denoting the hole, and the grammar $\mathfrak{C} ::= \Box \mid (\mathfrak{C}\ \mathfrak{T}) \mid (\mathfrak{T}\ \mathfrak{C})$ with the above rule for $\mathfrak{T}$ generates pre-terms containing exactly one occurrence of the hole. Given a type for the hole, a *context* is an element of $\mathfrak{C}$ which is typed as a term is. Let $C[]_A$ denote a context in which the hole has type A; filling the hole with a term $t : A$ produces the term $C[t]_A$ defined as follows: $\Box[t]_A = t$, $(C_0[]_A\ t_1)[t]_A = C_0[t]_A\ t_1$ and $(t_0\ C_1[]_A)[t]_A = t_0\ C_1[t]_A$. We usually omit types in the above notation, and in $C[t]$, t is understood as a term which has the same type as the hole does.

Rules and reduction Now we have all the ingredients in our recipe for higher-order term rewriting. A *rewrite rule* $\ell \to r$ is an ordered pair of terms where ℓ and r have the same type, $\mathrm{Var}(\ell) \supseteq \mathrm{Var}(r)$ and ℓ assumes the form $f\ t_1 \cdots t_n$ for some function symbol f. Formally, a *simply-typed term rewriting system* (STRS) is a quadruple $(\mathcal{S}, \mathcal{F}, \mathcal{V}, \mathcal{R})$ where every element of $\mathcal{F} \cup \mathcal{V}$ is assigned a simple type over $\mathcal{S}$ and $\mathcal{R} \subseteq T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V})$ is a set of rewrite rules. We usually let $\mathcal{R}$ alone stand for the STRS and keep the details of term formation implicit.

The set $\mathcal{R}$ of rewrite rules induces the *rewrite relation* $\to_{\mathcal{R}} \subseteq T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V})$: $t \to_{\mathcal{R}} t'$ if and only if there exist a rewrite rule $\ell \to r \in \mathcal{R}$, a substitution σ and a context $C[]$ such that $t = C[\ell\sigma]$ and $t' = C[r\sigma]$. When there is no ambiguity about the STRS in question, we may simply write $\to$ for $\to_{\mathcal{R}}$.

Given a relation $\succ \subseteq X \times X$, an element x of X is called *terminating* with respect to $\succ$ if there is no infinite sequence $x = x_0 \succ x_1 \succ \cdots$, and $\succ$ is called *well-founded* if all the elements of X are terminating with respect to $\succ$. An STRS $\mathcal{R}$ is called terminating if $\to_{\mathcal{R}}$ is well-founded.

Example 1. The following rewrite rules constitute a terminating STRS:

$$\mathsf{take}\ \mathsf{zero}\ l \to \mathsf{nil} \quad \mathsf{take}\ n\ \mathsf{nil} \to \mathsf{nil} \quad \mathsf{take}\ (\mathsf{suc}\ n)\ (\mathsf{cons}\ x\ l) \to \mathsf{cons}\ x\ (\mathsf{take}\ n\ l)$$

where $\mathsf{zero} : \mathsf{nat}$, $\mathsf{suc} : \mathsf{nat} \to \mathsf{nat}$, $\mathsf{nil} : \mathsf{natlist}$, $\mathsf{cons} : \mathsf{nat} \to \mathsf{natlist} \to \mathsf{natlist}$ and $\mathsf{take} : \mathsf{nat} \to \mathsf{natlist} \to \mathsf{natlist}$ are function symbols, and $l : \mathsf{natlist}$, $n : \mathsf{nat}$ and $x : \mathsf{nat}$ are variables.

Example 2. The following rewrite rule constitutes a non-terminating STRS:

$$\mathsf{iterate}\ f\ x \to \mathsf{cons}\ x\ (\mathsf{iterate}\ f\ (f\ x))$$

where $\mathsf{cons} : \mathsf{nat} \to \mathsf{natlist} \to \mathsf{natlist}$ and $\mathsf{iterate} : (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat} \to \mathsf{natlist}$ are function symbols, and $f : \mathsf{nat} \to \mathsf{nat}$ and $x : \mathsf{nat}$ are variables.

Limitations. The formalism presented above does not offer product types, polymorphism or λ-abstractions. What it does offer is its already expressive syntax enabling us, in a higher-order setting, to generalize LCTRSs and to discover what challenges one may face when extending existing unconstrained techniques. We expect that, once preliminary higher-order results are developed, we will adopt more features from other unconstrained higher-order formats in future extensions of LCSTRSs.

The exclusion of λ-abstractions does not rid us of first-class functions, thanks to curried notation. For example, the occurrence of suc in $\mathsf{iterate}\ \mathsf{suc}\ \mathsf{zero}$ is partially (in this case, not at all) applied and still forms a term, which can be passed as an argument. Also, a term such as $\mathsf{iterate}\ (\lambda x.\, \mathsf{suc}\ (\mathsf{suc}\ x))\ \mathsf{zero}$ can be simulated at the cost of an extra rewrite rule (in this case, $\mathsf{add2}\ x \to \mathsf{suc}\ (\mathsf{suc}\ x)$). There are also straightforward ways to encode product types.

Notions of termination. If we combine the two STRSs from Examples 1 and 2, the outcome is surely non-terminating: take zero (iterate suc zero) is not terminating, for example. From a Haskell programmer's perspective, however, this term is "terminating" due to the non-strictness of Haskell. In general, every functional language uses a certain evaluation strategy to choose a specific redex, if any, to rewrite within a term, whereas the rewrite relation we define in this section corresponds to full rewriting: the redex is chosen non-deterministically.

Furthermore, programmers usually care only about the termination of terms that are reachable from the entry point of a program and seldom consider full termination: the termination of all terms, i.e., the well-foundedness of the rewrite relation. We study full termination with respect to full rewriting in this paper, as it implies any other termination properties and full termination is often a prerequisite for determining properties such as confluence and equivalence.

3 Logically Constrained STRSs

Term rewriting systems do not have in-built types; they rely on inductively defined data types such as unary numbers. While this makes them highly suited to inductive reasoning, it means concepts like integers or arrays need to be encoded. This can be quite impractical; for example, the number 1023 would be encoded by a term suc (suc ... (suc zero)...) of 1024 symbols, and reasoning with negative numbers quickly introduces complications. Moreover, such reasoning cannot take advantage of the advances in the SMT-community.

Hence, in this section, we introduce logical constraints into STRSs so that data types that are not (co)inductively defined—like integers, bitvectors and arrays—can be represented directly, and implementations of analysis techniques can use existing SMT-solvers. We follow the ideas for first-order LCTRSs in [24,12], but use STRSs as a starting point. Specifically, we will allow systems over *arbitrary first-order theories* (so we do not limit interest to, e.g., systems over integers, but do restrict our theories to avoid higher-order constraints), using partial application and higher-order variables for the unconstrained part.

3.1 Terms Modulo Theories

Following Section 2, we postulate a set $\mathcal{S}$ of sorts, a set $\mathcal{F}$ of function symbols and a set $\mathcal{V}$ of variables where every element of $\mathcal{F} \cup \mathcal{V}$ is assigned a simple type over $\mathcal{S}$. First, we assume that there is a distinguished subset $\mathcal{S}_\vartheta$ of $\mathcal{S}$, called the set of *theory sorts*. The grammar $\mathcal{T}_\vartheta ::= \mathcal{S}_\vartheta \mid (\mathcal{S}_\vartheta \to \mathcal{T}_\vartheta)$ generates the set $\mathcal{T}_\vartheta$ of *theory types* over $\mathcal{S}_\vartheta$. Next, we assume that there is a distinguished subset $\mathcal{F}_\vartheta$ of $\mathcal{F}$, called the set of *theory symbols*, and that the type of every theory symbol is in $\mathcal{T}_\vartheta$, which means that the type of any argument passed to a theory symbol is a theory sort. Elements of $T(\mathcal{F}_\vartheta, \mathcal{V})$ are called *theory terms*. Last, for technical reasons, we assume that there are infinitely many variables of each type.

Theory symbols are interpreted in an underlying theory: given an $\mathcal{S}_\vartheta$-indexed family of sets $(\mathfrak{X}_A)_{A \in \mathcal{S}_\vartheta}$, we extend it to a $\mathcal{T}_\vartheta$-indexed family by letting $\mathfrak{X}_{A \to B}$

be the set of mappings from $\mathfrak{X}_A$ to $\mathfrak{X}_B$; an *interpretation* of theory symbols is a $\mathcal{T}_\vartheta$-indexed family of mappings $(\llbracket\cdot\rrbracket_A)_{A\in\mathcal{T}_\vartheta}$ where $\llbracket\cdot\rrbracket_A$ assigns to each theory symbol which has type A an element of $\mathfrak{X}_A$ and is bijective if $A \in \mathcal{S}_\vartheta$. Theory symbols whose type is a theory sort are called *values*. Given an interpretation of theory symbols $(\llbracket\cdot\rrbracket_A)_{A\in\mathcal{T}_\vartheta}$, we extend each indexed mapping $\llbracket\cdot\rrbracket_B$ to one that assigns to each *ground theory term* of type B an element of $\mathfrak{X}_B$ by letting $\llbracket t_0\ t_1\rrbracket_B$ be $\llbracket t_0\rrbracket_{A\to B}(\llbracket t_1\rrbracket_A)$. We usually write just $\llbracket\cdot\rrbracket$ when the type can be deduced.

Example 3. Let $\mathcal{S}_\vartheta$ be $\{\,\mathsf{int}\,\}$. Then $\mathsf{int}\to\mathsf{int}\to\mathsf{int}$ is a theory type over $\mathcal{S}_\vartheta$ while $(\mathsf{int}\to\mathsf{int})\to\mathsf{int}$ is not. Let $\mathcal{F}_\vartheta$ be $\{\,\mathsf{sub}\,\}\cup\{\,\bar{n}\mid n\in\mathbb{Z}\,\}$ where $\mathsf{sub}:\mathsf{int}\to\mathsf{int}\to\mathsf{int}$ and $\bar{n}:\mathsf{int}$. The values are the elements of $\{\,\bar{n}\mid n\in\mathbb{Z}\,\}$. Let $\mathfrak{X}_{\mathsf{int}}$ be $\mathbb{Z}$, $\llbracket\cdot\rrbracket_{\mathsf{int}}$ be the mapping $\bar{n}\mapsto n$ and $\llbracket\mathsf{sub}\rrbracket$ be the mapping $\lambda m.\,\lambda n.\,m-n$. The interpretation of $\mathsf{sub}\ \bar{1}$ is the mapping $\lambda n.\,1-n$.

We are not limited to the theory of integers:

Example 4. To reason about programs with integer arrays, we could either encode arrays as lists (and implement storing and reading from lists through recursion), or consider a theory of bounded integer arrays. For the latter, let $\mathcal{S}_\vartheta$ be $\{\,\mathsf{int},\mathsf{intarray}\,\}$ and $\mathcal{F}_\vartheta$ the union of $\{\,\mathsf{size},\mathsf{select},\mathsf{store}\,\}$, $\{\,\bar{n}\mid n\in\mathbb{Z}\,\}$ and $\{\,\langle\bar{n}_0,\ldots,\bar{n}_{k-1}\rangle\mid k\in\mathbb{N}\text{ and }\forall i.\,n_i\in\mathbb{Z}\,\}$ where $\mathsf{size}:\mathsf{intarray}\to\mathsf{int}$, $\mathsf{select}:\mathsf{intarray}\to\mathsf{int}\to\mathsf{int}$, $\mathsf{store}:\mathsf{intarray}\to\mathsf{int}\to\mathsf{int}\to\mathsf{intarray}$, $\bar{n}:\mathsf{int}$ and $\langle\bar{n}_0,\ldots,\bar{n}_{k-1}\rangle:\mathsf{intarray}$. Let $\mathfrak{X}_{\mathsf{int}}$ and $\mathfrak{X}_{\mathsf{intarray}}$ be $\mathbb{Z}$ and $\mathbb{Z}^*$, respectively. Let $\llbracket\cdot\rrbracket_{\mathsf{int}}$ be the mapping $\bar{n}\mapsto n$ and $\llbracket\cdot\rrbracket_{\mathsf{intarray}}$ the mapping $\langle\bar{n}_0,\ldots,\bar{n}_{k-1}\rangle\mapsto n_0\ldots n_{k-1}$. Let $\llbracket\mathsf{size}\rrbracket(n_0\ldots n_{k-1})$ be k. Let $\llbracket\mathsf{select}\rrbracket(n_0\ldots n_{k-1},i)$ be n_i if $0\le i<k$, and 0 otherwise. Let $\llbracket\mathsf{store}\rrbracket(n_0\ldots n_{k-1},i,m)$ be $n_0\ldots n_{i-1}mn_{i+1}\ldots n_{k-1}$ if $0\le i<k$, and $n_0\ldots n_{k-1}$ otherwise. Then the values are the symbols $\bar{n}:\mathsf{int}$ and $\langle\bar{n}_0,\ldots,\bar{n}_{k-1}\rangle:\mathsf{intarray}$.

In this paper we largely use the integer theory from Example 3 in examples because it is easily understood. When considering program analysis, the alternative theory of *bitvectors* may be more appropriate, since in practice, unbounded integers are rarely considered. Note also that we do not limit interest to the theories in SMT-LIB [7]. For example, the theory of integer arrays in Example 4 does not exactly match a pre-defined theory (although it could be encoded within the theory of functional arrays). It *is* useful to limit interest to theories supported by SMT-solvers when considering automation.

3.2 Constrained Rewriting

Constrained rewriting requires the theory sort bool: we henceforth assume that $\mathsf{bool}\in\mathcal{S}_\vartheta$, $\{\,\mathfrak{f},\mathfrak{t}\,\}\subseteq\mathcal{F}_\vartheta$, $\mathfrak{X}_{\mathsf{bool}}=\{\,0,1\,\}$, $\llbracket\mathfrak{f}\rrbracket_{\mathsf{bool}}=0$ and $\llbracket\mathfrak{t}\rrbracket_{\mathsf{bool}}=1$. A *logical constraint* is a theory term φ such that φ has type bool and the type of each variable in $\mathrm{Var}(\varphi)$ is a theory sort. A (constrained) *rewrite rule* is a triple $\ell\to r\ [\varphi]$ where ℓ and r are terms which have the same type, φ is a logical constraint, the type of each variable in $\mathrm{Var}(r)\setminus\mathrm{Var}(\ell)$ is a theory sort and ℓ is not a theory term while assuming the form $f\ t_1\cdots t_n$ for some function symbol f.

This definition can be obscure at first glance, especially when compared with its unconstrained counterpart in Section 2: variables which do not occur in ℓ are allowed to occur in r, not to mention the logical constraint φ as a brand-new component. Given a rewrite rule $\ell \to r\ [\varphi]$, the idea is that variables occurring in φ are to be instantiated to values which make φ true and other variables which occur in r but not in ℓ are to be instantiated to arbitrary values—note that the type of each of these variables is a theory sort. Formally, given an interpretation of theory symbols $[\![\cdot]\!]$, a substitution σ is said to *respect* a rewrite rule $\ell \to r\ [\varphi]$ if $\sigma(x)$ is a value for all $x \in \mathrm{Var}(\varphi) \cup (\mathrm{Var}(r) \setminus \mathrm{Var}(\ell))$ and $[\![\varphi\sigma]\!] = 1$.

We summarize all the above ingredients in the following definition:

Definition 1. *A* logically constrained STRS *(LCSTRS) consists of $\mathcal{S}$, $\mathcal{S}_\vartheta$, $\mathcal{F}$, $\mathcal{F}_\vartheta$, $\mathcal{V}$, $(\mathfrak{X}_A)$, $[\![\cdot]\!]$ and $\mathcal{R}$ where*

1. *$\mathcal{S}$ is a set of sorts,*
2. *$\mathcal{S}_\vartheta \subseteq \mathcal{S}$ is a set of theory sorts which contains* bool*,*
3. *$\mathcal{F}$ is a set of function symbols in which every function symbol is assigned a simple type over $\mathcal{S}$,*
4. *$\mathcal{F}_\vartheta \subseteq \mathcal{F}$ is a set of theory symbols in which the type of every theory symbol is a theory type over $\mathcal{S}_\vartheta$, with $\mathfrak{f} : \mathsf{bool}$ and $\mathfrak{t} : \mathsf{bool}$ elements of $\mathcal{F}_\vartheta$,*
5. *$\mathcal{V}$ is a set of variables disjoint from $\mathcal{F}$ in which every variable is assigned a simple type over $\mathcal{S}$ and there are infinitely many variables to which every type is assigned,*
6. *$(\mathfrak{X}_A)$ is an $\mathcal{S}_\vartheta$-indexed family of sets such that $\mathfrak{X}_{\mathsf{bool}} = \{0, 1\}$,*
7. *$[\![\cdot]\!]$ is an interpretation of theory symbols such that $[\![\mathfrak{f}]\!] = 0$ and $[\![\mathfrak{t}]\!] = 1$, and*
8. *$\mathcal{R} \subseteq T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}_\vartheta, \mathcal{V})$ is a set of rewrite rules.*

We usually let $\mathcal{R}$ alone stand for the LCSTRS.

And the following definition concludes the elaboration of constrained rewriting:

Definition 2. *Given an LCSTRS $\mathcal{R}$, the set of rewrite rules induces the* rewrite relation *$\to_\mathcal{R} \subseteq T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V})$ such that $t \to_\mathcal{R} t'$ if and only if one of the following conditions is true:*

1. *There exist a rewrite rule $\ell \to r\ [\varphi] \in \mathcal{R}$, a substitution σ which respects $\ell \to r\ [\varphi]$ and a context $C[]$ such that $t = C[\ell\sigma]$ and $t' = C[r\sigma]$.*
2. *There exist theory symbols $v_1 : A_1, \ldots, v_n : A_n, v' : B$ and $f : A_1 \to \cdots \to A_n \to B$ for $n > 0$ and $A_1, \ldots, A_n, B \in \mathcal{S}_\vartheta$ such that $[\![f\ v_1 \cdots v_n]\!] = [\![v']\!]$, and a context $C[]$ such that $t = C[f\ v_1 \cdots v_n]$ and $t' = C[v']$.*

Note that the above conditions cannot be satisfied for the same context $C[]$: $f\ v_1 \cdots v_n$ is a theory term, whereas ℓ is not a theory term in any rule $\ell \to r\ [\varphi]$. If $t \to_\mathcal{R} t'$ due to the second condition, we also write $t \to_\kappa t'$ and call it a step of calculation*. When no ambiguity arises, we may simply write $\to$ for $\to_\mathcal{R}$.*

Example 5. We can rework the STRS from Example 1 into an LCSTRS:

$$\begin{array}{lll} \mathsf{take}\ n\ l \to \mathsf{nil} & [n \leq 0] & \mathsf{take}\ n\ \mathsf{nil} \to \mathsf{nil} \\ \mathsf{take}\ n\ (\mathsf{cons}\ x\ l) \to \mathsf{cons}\ x\ (\mathsf{take}\ (n-1)\ l) & [n > 0] & \end{array}$$

where $\mathcal{S} = \mathcal{S}_\vartheta \cup \{ \mathsf{intlist} \}$, $\mathcal{S}_\vartheta = \{ \mathsf{bool}, \mathsf{int} \}$, $\mathcal{F} = \mathcal{F}_\vartheta \cup \{ \mathsf{nil}, \mathsf{cons}, \mathsf{take} \}$, $\mathcal{F}_\vartheta = \{ \leq, >, -, \mathfrak{f}, \mathfrak{t} \} \cup \mathbb{Z}$, $\mathcal{V} \supseteq \{ l, n, x \}$, $\leq : \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$, $> : \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$, $- : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$, $\mathfrak{f} : \mathsf{bool}$, $\mathfrak{t} : \mathsf{bool}$, $v : \mathsf{int}$ for all $v \in \mathbb{Z}$, $\mathsf{nil} : \mathsf{intlist}$, $\mathsf{cons} : \mathsf{int} \to \mathsf{intlist} \to \mathsf{intlist}$, $\mathsf{take} : \mathsf{int} \to \mathsf{intlist} \to \mathsf{intlist}$, $l : \mathsf{intlist}$, $n : \mathsf{int}$ and $x : \mathsf{int}$.

Here and henceforth we let integer literals and operators, e.g., 0, 1, $\leq$, $>$ and $-$, denote both the corresponding theory symbols and their respective images under the interpretation—in contrast to Examples 3 and 4, where we pedantically make a distinction between, say, $\bar{1}$ and 1. We also use infix notation for some binary operators to improve readability, and omit the logical constraint of a rewrite rule when it is $\mathfrak{t}$. Below is a rewrite sequence:

$$\begin{aligned} &\mathsf{take}\ 1\ (\mathsf{cons}\ x\ (\mathsf{cons}\ y\ l)) \to \mathsf{cons}\ x\ (\mathsf{take}\ (1-1)\ (\mathsf{cons}\ y\ l)) \\ \to_\kappa\ &\mathsf{cons}\ x\ (\mathsf{take}\ 0\ (\mathsf{cons}\ y\ l)) \to \mathsf{cons}\ x\ \mathsf{nil} \end{aligned}$$

Example 6. In Section 1, the rewrite rules implementing the factorial function in continuation-passing style constitute an LCSTRS. Below is a rewrite sequence:

$$\begin{aligned} \mathsf{fact}\ 1\ k &\to \mathsf{fact}\ (1-1)\ (\mathsf{comp}\ k\ ((*)\ 1)) \to_\kappa \mathsf{fact}\ 0\ (\mathsf{comp}\ k\ ((*)\ 1)) \\ &\to \mathsf{comp}\ k\ ((*)\ 1)\ 1 \to k\ ((*)\ 1\ 1) \to_\kappa k\ 1 \end{aligned}$$

Example 7. Consider a rewrite rule $\mathsf{readint} \to n$ with a variable $n : \mathsf{int}$ that occurs on the right-hand side of $\to$ but not on the left. This is not allowed in STRSs, but is in LCSTRSs. It looks as if we might rewrite $\mathsf{readint}$ to a variable, but this is not the case: all the substitutions which respect this rewrite rule must map n to a value. Indeed, $\mathsf{readint}$ is always rewritten to a value of type int. We may have, say, $\mathsf{readint} \to 42$. Such variables can be used to model user input.

Example 8. Getting input by means of the rewrite rule from Example 7 has one flaw: in case of multiple integers to be read, the order of reading each is nondeterministic. Even in the presence of an evaluation strategy, the order may not be the desired one. We can use continuation-passing style to solve this problem:

$$\mathsf{readint}\ k \to k\ n \quad \mathsf{comp}\ g\ f\ x \to g\ (f\ x) \quad \mathsf{sub} \to \mathsf{readint}\ (\mathsf{comp}\ \mathsf{readint}\ (-))$$

where $\mathsf{comp} : ((\mathsf{int} \to \mathsf{int}) \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int}$. If the first and the second integers to be read were 1 and 2, respectively, the following rewrite sequence would be the only one starting from sub:

$$\begin{aligned} \mathsf{sub} &\to \mathsf{readint}\ (\mathsf{comp}\ \mathsf{readint}\ (-)) \to \mathsf{comp}\ \mathsf{readint}\ (-)\ 1 \\ &\to \mathsf{readint}\ ((-)\ 1) \to (-)\ 1\ 2 \to_\kappa -1 \end{aligned}$$

Since there is no way to specify the actual input within an LCSTRS, rewrite sequences such as the one above cannot be derived deterministically. Nevertheless, this example and Example 6 demonstrate that LCSTRSs can represent relatively sophisticated control mechanisms utilized by functional programs.

Remarks. We reflect on some of the concepts presented in this section:

- We use the phrase "terms modulo theories" in line with "satisfiability modulo theories": some function symbols are interpreted within a theory. While such an interpretation gives rise to a way of identifying certain terms, namely those that are convertible to each other with respect to $\to_\kappa$, we do not consider them identified (in other words, modulo κ) in this paper.
- First-order LCSTRSs—where the type order of each function symbol is no greater than one, variables with a non-zero type order (i.e., higher-order variables), are excluded, and the type of both sides of a rewrite rule is always a sort—coincide with LCTRSs. The second implementation of the factorial function in Section 1 is an example of a first-order LCSTRS.
- Logical constraints are essentially first-order: the type order of a theory symbol cannot be greater than one while higher-order variables are excluded. This restriction rules out, for example, the following implementation:

  $$\begin{array}{lll} \mathsf{filter}\ f\ (\mathsf{cons}\ x\ l) \to \mathsf{cons}\ x\ (\mathsf{filter}\ f\ l) & [f\ x] & \mathsf{filter}\ f\ \mathsf{nil} \to \mathsf{nil} \\ \mathsf{filter}\ f\ (\mathsf{cons}\ x\ l) \to \mathsf{filter}\ f\ l & [\neg\ (f\ x)] & \end{array}$$

  Note that the filter function can actually be implemented in an LCSTRS; the problem is not the higher-order variable f itself but its occurrence in logical constraints. In this case, because the filter function is usually meant to be used in combination with "user-defined" predicates—which are function symbols defined by rewrite rules and therefore do not belong to the theories—it makes sense to disallow f from occurring in logical constraints. In general, we may encounter use cases for higher-order constraints; until then, we focus on first-order constraints, which are very common in functional programs.

4 A Constrained Higher-Order recursive path Ordering

Termination is an important aspect of program analysis and has been studied by the term rewriting community for decades. Not only is termination itself critical to the correctness of certain programs, but it also facilitates other analyses by admitting well-founded induction on terms.

In this section, we approach the termination problem of LCSTRSs and illustrate how existing termination techniques for higher-order term rewriting may be extended to LCSTRSs. Specifically, we wil adapt HORPO [20] to our setting. This is one of the oldest, yet still effective techniques for higher-order termination. HORPO can be used both as a stand-alone method for proving termination, and inside higher-order definitions of a dependency pair framework [1,33,11,22]. Hence, this definition offers a solid basis for use in a termination module of an analysis tool. We will discuss the automation of this method in Section 5.

A definition of a constrained RPO for first-order LCTRSs was provided in [24]. We take inspiration from this definition for its approach to theory terms, but formalize the ideas and add support for (higher) types and partial application.

4.1 HORPO, Unconstrained and Uncurried

We first recall HORPO in its original form. Note that the original definition is based on an unconstrained and uncurried format, and a thorough discussion on it is beyond the scope of this paper. The following presentation is mostly informal and only serves the purposes of comparison and inspiration.

We begin with two standard definitions:

Definition 3. *Given relations $\succsim$ and $\succ$ over X, the* generalized lexicographic ordering $\succ^{\mathfrak{l}} \subseteq X^* \times X^*$ *is induced as follows: $x_1 \dots x_m \succ^{\mathfrak{l}} y_1 \dots y_n$ if and only if there exists $k \leq \min(m, n)$ such that $x_i \succsim y_i$ for all $i < k$ and $x_k \succ y_k$.*

Definition 4. *Given relations $\succsim$ and $\succ$ over X, the* generalized multiset ordering $\succ^{\mathfrak{m}} \subseteq X^* \times X^*$ *is induced as follows: $x_1 \dots x_m \succ^{\mathfrak{m}} y_1 \dots y_n$ if and only if there exist a non-empty subset I of $\{ 1, \dots, m \}$ and a mapping π from $\{ 1, \dots, n \}$ to $\{ 1, \dots, m \}$ such that*

1. $\forall i \in I. \forall j \in \pi^{-1}(i). x_i \succ y_j$,
2. $\forall i \in \{ 1, \dots, m \} \setminus I. \forall j \in \pi^{-1}(i). x_i \succsim y_j$, *and*
3. $\forall i \in \{ 1, \dots, m \} \setminus I. \left|\pi^{-1}(i)\right| = 1$.

In the following definition of HORPO, when we refer to the above definitions, $\succsim$ is the *equality* over terms and $\succ$ is HORPO itself.

Given a well-founded ordering $\blacktriangleright \subseteq \mathcal{F} \times \mathcal{F}$, called the *precedence*, and a mapping $\mathfrak{s} : \mathcal{F} \to \{ \mathfrak{l}, \mathfrak{m} \}$, called the *status*, HORPO is a type-preserving relation $\succ$ such that $s \succ t$ if and only if one of the following conditions is true:

1. $s = f(s_1, \dots, s_m)$, $f \in \mathcal{F}$ and $\exists k. s_k \succeq t$.
2. $s = f(s_1, \dots, s_m)$, $f \in \mathcal{F}$, $t = @(\dots @(@(t_0, t_1), t_2) \dots, t_n)$ and $\forall i. s \succ t_i \vee \exists k. s_k \succeq t_i$.
3. $s = f(s_1, \dots, s_m)$, $t = g(t_1, \dots, t_n)$, $f \in \mathcal{F}$, $g \in \mathcal{F}$, $f \blacktriangleright g$ and $\forall i. s \succ t_i \vee \exists k. s_k \succeq t_i$.
4. $s = f(s_1, \dots, s_m)$, $t = f(t_1, \dots, t_m)$, $f \in \mathcal{F}$, $\mathfrak{s}(f) = \mathfrak{l}$, $s_1 \dots s_m \succ^{\mathfrak{l}} t_1 \dots t_m$ and $\forall i. s \succ t_i \vee \exists k. s_k \succeq t_i$.
5. $s = f(s_1, \dots, s_m)$, $t = f(t_1, \dots, t_m)$, $f \in \mathcal{F}$, $\mathfrak{s}(f) = \mathfrak{m}$ and $s_1 \dots s_m \succ^{\mathfrak{m}} t_1 \dots t_m$.
6. $s = @(s_0, s_1)$, $t = @(t_0, t_1)$ and $s_0 s_1 \succ^{\mathfrak{m}} t_0 t_1$.
7. $s = \lambda x. s_0$, $t = \lambda x. t_0$ and $s_0 \succ t_0$.

Here $\succeq$ denotes the reflexive closure of $\succ$.

We call this format uncurried because every function symbol has an arity, i.e., the number of arguments guaranteed for each occurrence of the function symbol in a term. This is indicated by the functional notation, $f(s_1, \dots, s_m)$ instead of $f\ s_1 \cdots s_m$. For example, if f has arity m, its occurrence in a term must take exactly m arguments; a term $f(s_1, \dots, s_{m-1})$ is not allowed. A function symbol's type (or more technically, its type declaration) can permit more arguments than its arity guarantees. Such an extra argument is supplied through the syntactic form $@(\cdot, \cdot)$. For example, if the same function symbol f is given

an extra argument s_{m+1}, we write $@(f(s_1, \ldots, s_m), s_{m+1})$ This syntactic form is also used to pass arguments to variables and λ-abstractions.

The difference between an uncurried and a curried format is more than a notational issue, and poses technical challenges to the extension of HORPO to LCSTRSs, as we will see subsequently in this section. Another source of challenges is, as one would expect, constrained rewriting.

4.2 Rule Removal

HORPO works as a reduction ordering $\succ$, which is a type-preserving, *stable* (i.e., $t \succ t'$ implies $t\sigma \succ t'\sigma$), *monotonic* (i.e., $t \succ t'$ implies $C[t] \succ C[t']$) and well-founded relation. Note that despite its name, HORPO is not necessarily transitive. If such a relation *orients* all the rewrite rules in $\mathcal{R}$ (i.e., $\ell \succ r$ for all $\ell \to r \in \mathcal{R}$), we can conclude that the rewrite relation $\to_{\mathcal{R}}$ is well-founded.

A similar strategy for LCSTRSs requires a few tweaks. First, stability should be tightly coupled with rule orientation because every rewrite rule of an LCSTRS is equipped with a logical constraint, which decides what substitutions are expected when the rewrite rule is applied. Second, the monotonicity requirement can be weakened because ℓ is never a theory term in a rewrite rule $\ell \to r\ [\varphi]$. Hence, we define as follows:

Definition 5. *A type-preserving relation $\Rightarrow \subseteq T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V})$ is said*

1. *to* stably orient *a rewrite rule $\ell \to r\ [\varphi]$ if $\ell\sigma \Rightarrow r\sigma$ for each substitution σ which respects the rewrite rule, and*
2. *to be* rule-monotonic *if $t \Rightarrow t'$ implies $C[t] \Rightarrow C[t']$ when $t \notin T(\mathcal{F}_\vartheta, \mathcal{V})$.*

Besides having rewrite rules stably oriented, we need to deal with calculation. It turns out to be unnecessary to search for a well-founded relation which includes $\to_\kappa$, given the following observation:

Lemma 1. *$\to_\kappa$ is well-founded.*

Proof. The term size strictly decreases through every step of calculation. □

We rather look for a type-preserving and well-founded relation $\succ$ which stably orients every rewrite rule, is rule-monotonic, and is *compatible* with $\to_\kappa$, i.e., $\to_\kappa ; \succ \subseteq \succ^+$ or $\succ ; \to_\kappa \subseteq \succ^+$. This strategy is an instance of *rule removal*:

Theorem 1. *Given an LCSTRS $\mathcal{R}$, the rewrite relation $\to_{\mathcal{R}}$ is well-founded if and only if there exist sets $\mathcal{R}_1$ and $\mathcal{R}_2$ such that $\to_{\mathcal{R}_1}$ is well-founded and $\mathcal{R}_1 \cup \mathcal{R}_2 = \mathcal{R}$, and type-preserving, rule-monotonic relations $\Rightarrow$ and $\succ$ such that*

1. *$\Rightarrow$ includes $\to_\kappa$ and stably orients every rewrite rule in $\mathcal{R}_1$,*
2. *$\succ$ is well-founded and stably orients every rewrite rule in $\mathcal{R}_2$, and*
3. *$\Rightarrow ; \succ \subseteq \succ^+$ or $\succ ; \Rightarrow \subseteq \succ^+$.*

Here $\to_{\mathcal{R}_1}$ assumes the same term formation and interpretation as $\to_{\mathcal{R}}$ does.

Proof. If $\to_{\mathcal{R}}$ is well-founded, take $\mathcal{R}_1 = \emptyset$, $\mathcal{R}_2 = \mathcal{R}$, $\Rightarrow = \to_\kappa$ and $\succ = \to_{\mathcal{R}}$. Note that $\to_\emptyset = \to_\kappa$ by definition.

Now assume given $\mathcal{R}_1, \mathcal{R}_2, \Rightarrow$ and $\succ$. Since $\to_\kappa \subseteq \Rightarrow$ and $\Rightarrow$ is rule-monotonic and stably orients every rewrite rule in $\mathcal{R}_1$, $\to_{\mathcal{R}_1} \subseteq \Rightarrow$ (since any instance of the left-hand side of a rule is not a theory term). So the compatibility of $\succ$ with $\Rightarrow$ implies its compatibility with $\to_{\mathcal{R}_1}$, which in turn implies the well-foundedness of $\to_{\mathcal{R}_1} \cup \succ$, given that both $\to_{\mathcal{R}_1}$ and $\succ$ are well-founded. Since $\mathcal{R}_1 \cup \mathcal{R}_2 = \mathcal{R}$ and $\succ$ is a rule-monotonic relation which stably orients every rewrite rule in $\mathcal{R}_2$, $\to_{\mathcal{R}} \subseteq \to_{\mathcal{R}_1} \cup \succ$. Hence, $\to_{\mathcal{R}}$ is well-founded. □

In a termination proof of $\mathcal{R}$, Theorem 1 allows us to remove rewrite rules that are in $\mathcal{R}_2$ from $\mathcal{R}$. If none of the rewrite rules are left after iterations of rule removal, the termination of the original LCSTRS can be concluded with Lemma 1.

4.3 HORPO for LCSTRSs

Before we adapt HORPO for LCSTRSs, we discuss how the theory may be handled. For this, let us consider the following LCSTRS.

$$\mathsf{rec}\ n\ x\ f \to x \quad [n \leq 0] \qquad \mathsf{rec}\ n\ x\ f \to f\ (n-1)\ (\mathsf{rec}\ (n-1)\ x\ f) \quad [n > 0]$$

where $\mathsf{rec} : \mathsf{int} \to \mathsf{int} \to (\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int}$. In the second rewrite rule, the left-hand side of $\to$ is $\mathsf{rec}\ n\ x\ f$ while the right-hand side has a subterm $\mathsf{rec}\ (n-1)\ x\ f$. It is natural to expect $n \succ n-1$ in the construction of HORPO. Note that this is impossible with respect to any recursive path ordering for unconstrained rewriting because n is a variable occurring in $n-1$: in unconstrained HORPO, we in fact have $\mathsf{rec}(-(n,1), x, f) \succ \mathsf{rec}(n, x, f)$. Hence, we must somehow take the logical constraint $n > 0$ into account.

The occurrence of n in the logical constraint ensures that n is instantiated to a value, say 42, when the rewrite rule is applied, and it is sensible to have $42 \succ 42 - 1$. Also, $n > 0$ guarantees that all the sequences of such descents are finite, i.e., the ordering $\lambda m.\, \lambda n.\, m > 0 \wedge m > n$, denoted by $\sqsupset$, is well-founded. Let $\varphi \models \varphi'$ denote, on the assumption that φ and φ' are logical constraints such that $\mathrm{Var}(\varphi) \supseteq \mathrm{Var}(\varphi')$, that $[\![\varphi'\sigma]\!] = 1$ for each substitution σ which respects φ. Then we have $n > 0 \models n \sqsupset n-1$. We thus would like to have $s \succ t$ if $\varphi \models s \sqsupset t$.

However, with the same ordering $\sqsupset$, we have both $m > 0 \wedge m > n \models m \sqsupset n$ and $n > 0 \wedge n > m \models n \sqsupset m$, while we cannot have both $m \succ n$ and $n \succ m$ without breaking well-foundedness. To resolve this issue, we split $\succ$ into a family of relations $(\succ_\varphi)$ indexed by logical constraints, and let $s \succ_\varphi t$ be true if $\varphi \models s \sqsupset t$. We also introduce a separate family of relations $(\succsim_\varphi)$ such that $s \succsim_\varphi t$ if $\varphi \models s \sqsupseteq t$ where $\sqsupseteq$ is the reflexive closure of $\sqsupset$. Note that hence $\succsim_\varphi$ is *not* the reflexive closure of $\succ_\varphi$; if it was, then even $n \succsim_{n \geq 1} 1$ would not be obtainable.

Now we have a family of pairs $(\succsim_\varphi, \succ_\varphi)$, which does not seem to suit rule removal; after all, the essential requirement is a fixed relation which is type-preserving, rule-monotonic, well-founded and at least compatible with $\to_\kappa$. When the definition of HORPO for LCSTRSs is fully presented, we will show that $\succ_{\mathfrak{t}}$ is such a relation and stably orients a rewrite rule $\ell \to r\ [\varphi]$ if $\ell \succ_\varphi r$.

The annotation φ of HORPO does not capture variables in $\mathrm{Var}(r) \setminus \mathrm{Var}(\ell)$, which also have a part to play in the decision of what substitutions are expected when $\ell \to r\ [\varphi]$ is applied. We may use a new annotation to accommodate these variables but there is a hack (which also appears in [32]): given a variable in $\mathrm{Var}(r) \setminus \mathrm{Var}(\ell)$, it can be harmlessly appended to φ, syntactically and without tampering with any interpretation. We henceforth assume that $\mathrm{Var}(r) \setminus \mathrm{Var}(\ell) \subseteq \mathrm{Var}(\varphi)$, for any rewrite rule $\ell \to r\ [\varphi]$. We also say that a substitution σ respects a *logical constraint* φ if $\sigma(x)$ is a value for all $x \in \mathrm{Var}(\varphi)$ and $[\![\varphi\sigma]\!] = 1$.

Before presenting constrained HORPO, we recall that in [20] all sorts collapse into one, and for example, $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$ and $\mathsf{int} \to \mathsf{intlist} \to \mathsf{intlist}$ are considered equal. The idea is that the original rewrite relation can be embedded in the single-sorted one, and if the latter is well-founded, so is the former. We follow this convention and henceforth compare types by their $\to$-structure only.

Below $\succ_\varphi^{\mathfrak{l}}$ and $\succ_\varphi^{\mathfrak{m}}$ are induced by $\succsim_\varphi$ and $\succ_\varphi$:

Definition 6. *Constrained HORPO depends on the following parameters:*

1. *The interpretation of theory symbols* $\sqsupset_A \colon A \to A \to \mathsf{bool}$ *for all* $A \in \mathcal{S}_\vartheta$ *such that* $[\![\sqsupset_A]\!]$ *is a well-founded ordering over* $\mathfrak{X}_A$*. The interpretation* $[\![\sqsupseteq_A]\!]$ *is assumed to be the reflexive closure of* $[\![\sqsupset_A]\!]$*. We usually write just* $\sqsupset$ *and* $\sqsupseteq$ *because sorts collapse. Consider* $[\![\sqsupset]\!]$ *the union* $\bigcup_{A \in \mathcal{S}_\vartheta} [\![\sqsupset_A]\!]$*, and* $[\![\sqsupseteq]\!]$ *likewise.*
2. *The* precedence $\blacktriangleright$*, a well-founded ordering over* $\mathcal{F}$ *such that* $f \blacktriangleright g$ *for all* $f \in \mathcal{F} \setminus \mathcal{F}_\vartheta$ *and* $g \in \mathcal{F}_\vartheta$*.*
3. *The* status $\mathfrak{s}$*, a mapping from* $\mathcal{F}$ *to* $\{\mathfrak{l}, \mathfrak{m}_2, \mathfrak{m}_3, \dots\}$*.*

The higher-order recursive path ordering *(HORPO) is a family of pairs of type-preserving relations* $(\succsim_\varphi, \succ_\varphi)$ *indexed by logical constraints and defined by the following conditions:*

1. $s \succsim_\varphi t$ *if and only if one of the following conditions is true:*
   (a) s *and* t *are theory terms whose type is a sort,* $\mathrm{Var}(s) \cup \mathrm{Var}(t) \subseteq \mathrm{Var}(\varphi)$ *and* $\varphi \models s \sqsupseteq t$*.*
   (b) $s \succ_\varphi t$*.*
   (c) $s \downarrow_\kappa t$*.*
   (d) s *is not a theory term,* $s = s_0\ s_1$*,* $t = t_0\ t_1$*,* $s_0 \succsim_\varphi t_0$ *and* $s_1 \succsim_\varphi t_1$*.*
2. $s \succ_\varphi t$ *if and only if one of the following conditions is true:*
   (a) s *and* t *are theory terms whose type is a sort,* $\mathrm{Var}(s) \cup \mathrm{Var}(t) \subseteq \mathrm{Var}(\varphi)$ *and* $\varphi \models s \sqsupset t$*.*
   (b) s *and* t *have equal types and* $s \rhd_\varphi t$*.*
   (c) s *is not a theory term,* $s = f\ s_1 \cdots s_n$ *for some* $f \in \mathcal{F}$*,* $t = f\ t_1 \cdots t_n$*,* $\forall i.\, s_i \succsim_\varphi t_i$ *and* $\exists k.\, s_k \succ_\varphi t_k$*.*
   (d) s *is not a theory term,* $s = x\ s_1 \cdots s_n$ *for some* $x \in \mathcal{V}$*,* $t = x\ t_1 \cdots t_n$*,* $\forall i.\, s_i \succsim_\varphi t_i$ *and* $\exists k.\, s_k \succ_\varphi t_k$*.*
3. $s \rhd_\varphi t$ *if and only if* s *is not a theory term,* $s = f\ s_1 \cdots s_m$ *for some* $f \in \mathcal{F}$ *and one of the following conditions is true:*
   (a) $\exists k.\, s_k \succsim_\varphi t$*.*
   (b) $t = t_0\ t_1 \cdots t_n$*,* $\forall i.\, s \rhd_\varphi t_i$*.*

(c) $t = g\ t_1 \cdots t_n$, $f \blacktriangleright g$, $\forall i.\, s \rhd_\varphi t_i$.
(d) $t = f\ t_1 \cdots t_n$, $\mathfrak{s}(f) = \mathfrak{l}$, $s_1 \dots s_m \succ^{\mathfrak{l}}_\varphi t_1 \dots t_n$, $\forall i.\, s \rhd_\varphi t_i$.
(e) $t = f\ t_1 \cdots t_n$, $\mathfrak{s}(f) = \mathfrak{m}_k$, $k \leq n$, $s_1 \dots s_{\min(m,k)} \succ^{\mathfrak{m}}_\varphi t_1 \dots t_k$, $\forall i.\, s \rhd_\varphi t_i$.
(f) t *is a value or a variable in* $\mathrm{Var}(\varphi)$.

Here $s \downarrow_\kappa t$ *if and only if there exists a term* r *such that* $s \to^*_\kappa r$ *and* $t \to^*_\kappa r$.

Differences to unconstrained HORPO. Conditions 1d, 2c and 2d are included in the definition so that $\succsim_\varphi$ and $\succ_\varphi$ are rule-monotonic. We stress that it is mandatory to use the weakened, rule-monotonicity requirement rather than the traditional monotonicity requirement: if $\succ_\varphi$ is monotonic, $1 \succ_\mathfrak{t} 0$ implies $1-1 \succ_\mathfrak{t} 1-0$, but $\mathfrak{t} \models (1-0) \sqsupset (1-1)$, i.e., $\succ_\mathfrak{t}$ cannot possibly be well-founded. This is why several rules have a limitation that s is not a theory term.

From curried notation, another issue related to rule-monotonicity arises, which leads to the above definition of $\rhd_\varphi$. If we had the original HORPO naively mirrored, the definition of $\succ_\varphi$ would include a condition which corresponds to condition 3b and reads: "$s \succ_\varphi t$ if s is not a theory term, $s = f\ s_1 \cdots s_m$ for some $f \in \mathcal{F}$, $t = t_0\ t_1 \cdots t_n$ and $\forall i.\, s \succ_\varphi t_i \vee \exists k.\, s_k \succsim_\varphi t_i$". Assume given such terms s and t, and that, say, $s \succ_\varphi t_1$. Now if there is a term r to which s can be applied, we have a problem with proving $s\ r \succ_\varphi t\ r = t_0\ t_1 \cdots t_n\ r$ because $s\ r \succ_\varphi t_1$ is not obtainable due to the type restriction. Note that $\succsim_\varphi$ and $\succ_\varphi$ are by definition type-preserving, whereas $\rhd_\varphi$ is not. We use $\rhd_\varphi$ to overcome this limitation. This lowering of the type restriction actually makes the definition significantly more powerful, and is reminiscent of the distinction between $\succ$ and $\succ_{\mathcal{T}_\mathcal{S}}$ in later versions of HORPO (e.g., [4]). However, we do not yet include other extensions from these works, and except for the type restriction and uncurried notation, the conditions of $\rhd_\varphi$ largely match those of the original HORPO.

Another subtle difference is the use of generalized lexicographic and multiset extensions: in the original HORPO, $\succsim$ is the reflexive closure of $\succ$ and therefore the traditional definitions of lexicographic and multiset extension suffice. Here, as observed before, this would be needlessly restrictive.

The split of a single multiset status label in $\mathfrak{m}_2, \mathfrak{m}_3, \dots$ is due to curried notation; in particular the possibility of partial application. If we had only a single multiset status which would for instance allow both $f\ 2\ 2 \rhd_\mathfrak{t} f\ 1$ and $f\ 1\ 3 \succ_\mathfrak{t} f\ 2\ 2$ to hold, then we would not have a well-founded ordering: if also $f \blacktriangleright g$ we would then have $f\ 2\ 2 \succ_\mathfrak{t} g\ (f\ 1)$ by 2b,3c, and $g\ (f\ 1) \succ_\mathfrak{t} f\ 1\ 3$ by 2b,3b. This change, also, adds some power to constrained HORPO: we can for instance prove termination of a system $\mathsf{f}\ x\ \mathsf{a}\ y \to \mathsf{f}\ \mathsf{b}\ x\ (\mathsf{g}\ y)$, by choosing $\mathfrak{s}(\mathsf{f}) = \mathfrak{m}_2$, which we could not orient if—like in the traditional HORPO definition—all arguments must be considered. The reason we do not consider a status $\mathfrak{m}_1$ is that this case is already covered by assigning a status $\mathfrak{l}$.

Given an LCSTRS $\mathcal{R}$, if we can divide the set of rules into two subsets $\mathcal{R}_1$ and $\mathcal{R}_2$, and find a combination of $[\![\sqsupset]\!]$, $\blacktriangleright$ and $\mathfrak{s}$ that guarantees $\ell \succsim_\varphi r$ for all $\ell \to r\ [\varphi] \in \mathcal{R}_1$ and $\ell \succ_\varphi r$ for all $\ell \to r\ [\varphi] \in \mathcal{R}_2$, the termination of $\mathcal{R}$ is reduced to that of $\mathcal{R}_1$. Before proving the soundness, we check out some examples:

Example 9. We continue the analysis of the motivating example rec. Let $[\![\sqsupset_{\mathsf{int}}]\!]$ be $\lambda m.\, \lambda n.\, m > 0 \wedge m > n$ as above. There is only one function symbol in $\mathcal{F} \setminus \mathcal{F}_\vartheta$, and it turns out that $\blacktriangleright$ can be any precedence. Let $\mathfrak{s}$ be a mapping such that $\mathfrak{s}(\mathsf{rec}) = \mathfrak{l}$. The first rewrite rule can be removed due to conditions 2b and 3a. The second rewrite rule can be removed as follows:

1. $\mathsf{rec}\ n\ x\ f \succ_{n>0} f\ (n-1)\ (\mathsf{rec}\ (n-1)\ x\ f)$ by 2b, 2.
2. $\mathsf{rec}\ n\ x\ f \rhd_{n>0} f\ (n-1)\ (\mathsf{rec}\ (n-1)\ x\ f)$ by 3b, 3, 4, 5.
3. $\mathsf{rec}\ n\ x\ f \rhd_{n>0} f$ by 3a, 6.
4. $\mathsf{rec}\ n\ x\ f \rhd_{n>0} n-1$ by 3a, 7.
5. $\mathsf{rec}\ n\ x\ f \rhd_{n>0} \mathsf{rec}\ (n-1)\ x\ f$ by 3d, 8, 4, 9, 3.
6. $f \succsim_{n>0} f$ by 1c.
7. $n \succsim_{n>0} n-1$ by 1a.
8. $n \succ_{n>0} n-1$ by 2a.
9. $\mathsf{rec}\ n\ x\ f \rhd_{n>0} x$ by 3a, 10.
10. $x \succsim_{n>0} x$ by 1c.

Example 10. Consider the LCSTRS from Example 5. Let $[\![\sqsupset_{\mathsf{int}}]\!]$ be $\lambda m.\, \lambda n.\, m > 0 \wedge m > n$. Let $\blacktriangleright$ be a precedence such that $\mathsf{take} \blacktriangleright \mathsf{nil}$ and $\mathsf{take} \blacktriangleright \mathsf{cons}$. Let $\mathfrak{s}$ be a mapping such that $\mathfrak{s}(\mathsf{take}) = \mathfrak{l}$. Then we can remove all of the rewrite rules. Note that to establish $\mathsf{take}\ n\ (\mathsf{cons}\ x\ l) \succ_{n>0} \mathsf{cons}\ x\ (\mathsf{take}\ (n-1)\ l)$, we need $\mathsf{cons}\ x\ l \succsim_{n>0} x$, which is obtainable because $\mathsf{intlist}$ is not distinguished from int.

4.4 Properties of Constrained HORPO

The soundness of constrained HORPO as a technique for rule removal relies on the following properties, which we now prove.

Rule orientation. The goal consists of two parts: $\succsim_{\mathfrak{t}}$ stably orients a rewrite rule $\ell \to r\ [\varphi]$ if $\ell \succsim_\varphi r$, and $\succ_{\mathfrak{t}}$ stably orients a rewrite rule $\ell \to r\ [\varphi]$ if $\ell \succ_\varphi r$. The core of the argument is to prove the following lemma:

Lemma 2. *Given logical constraints φ and φ' such that $\mathrm{Var}(\varphi) \supseteq \mathrm{Var}(\varphi')$ and $\varphi \models \varphi'$, then $\mathfrak{t} \models \varphi'\sigma$ for each substitution σ which respects φ.*

Proof. It follows from $\varphi \models \varphi'$ that $[\![\varphi'\sigma]\!] = 1$. Note that $\mathrm{Var}(\varphi'\sigma) = \emptyset$, and therefore $\varphi'\sigma\sigma' = \varphi'\sigma$ for all σ'. Hence, $\mathfrak{t} \models \varphi'\sigma$. □

And the rest is routine:

Theorem 2. *Given a logical constraint φ, terms s and t, the following statements are true for each substitution σ which respects φ:*

1. *$s \succsim_\varphi t$ implies $s\sigma \succsim_{\mathfrak{t}} t\sigma$.*
2. *$s \succ_\varphi t$ implies $s\sigma \succ_{\mathfrak{t}} t\sigma$.*
3. *$s \rhd_\varphi t$ implies $s\sigma \rhd_{\mathfrak{t}} t\sigma$.*

Proof. By mutual induction on the derivation. Note that $\to_\kappa$ is stable. □

Rule-monotonicity. Both $\succsim_\varphi$ and $\succ_\varphi$ are rule-monotonic for all φ. The former is trivial to prove, and the key to proving the latter is the following lemma:

Lemma 3. *$f\ s_1 \cdots s_m\ r \rhd_\varphi t$ if $f\ s_1 \cdots s_m \rhd_\varphi t$.*

Proof. By induction on the derivation. □

Now we can prove the rule-monotonicity:

Theorem 3. *$\succ_\varphi$ is rule-monotonic.*

Proof. By induction on the context $C[]$. Essentially, we ought to prove that given terms s and t which have equal types, if s is not a theory term and $s \succ_\varphi t$, $s\ r \succ_\varphi t\ r$ for all r, and $r\ s \succ_\varphi r\ t$ for all r. We prove the former by case analysis on the derivation of $s \succ_\varphi t$, and prove the latter by case analysis on r: $r = f\ r_1 \cdots r_n$ for some $f \in \mathcal{F}$ or $r = x\ r_1 \cdots r_n$ for some $x \in \mathcal{V}$. □

Compatibility. The strict relation $\succ_\mathfrak{t}$ is compatible with its non-strict counterpart $\succsim_\mathfrak{t}$; we prove that $\succsim_\mathfrak{t} ; \succ_\mathfrak{t} \subseteq \succ_\mathfrak{t} \cup (\succ_\mathfrak{t} ; \succ_\mathfrak{t})$, given the following observation:

Theorem 4. $\succsim_\mathfrak{t} = \succ_\mathfrak{t} \cup \downarrow_\kappa$.

Proof. By definition, $\succsim_\mathfrak{t} \supseteq \succ_\mathfrak{t} \cup \downarrow_\kappa$. We prove $\succsim_\mathfrak{t} \subseteq \succ_\mathfrak{t} \cup \downarrow_\kappa$ by induction on the derivation of $s \succsim_\mathfrak{t} t$. Only two cases are non-trivial. If s and t are ground theory terms whose type is a sort and $[\![s \sqsupseteq t]\!] = 1$, we have either $[\![s \sqsupset t]\!] = 1$ or $[\![s]\!] = [\![t]\!]$, and the former implies $s \succ_\mathfrak{t} t$ while the latter implies $s \downarrow_\kappa t$. On the other hand, if s is not a theory term, $s = s_0\ s_1$, $t = t_0\ t_1$, $s_0 \succsim_\mathfrak{t} t_0$ and $s_1 \succsim_\mathfrak{t} t_1$, by induction, if $s_0 \succ_\mathfrak{t} t_0$ or $s_1 \succ_\mathfrak{t} t_1$, we can prove $s \succ_\mathfrak{t} t$ in the same manner as we prove the rule-monotonicity of $\succ_\mathfrak{t}$, or $s_0 \downarrow_\kappa t_0$ and $s_1 \downarrow_\kappa t_1$, then $s \downarrow_\kappa t$. □

It remains to prove that $\downarrow_\kappa ; \succ_\mathfrak{t} \subseteq \succ_\mathfrak{t}$, which is implied by the following lemma:

Lemma 4. *Given terms s and s' such that $s \to_\kappa s'$, the following statements are true for all t:*

1. *$s \succsim_\mathfrak{t} t$ if and only if $s' \succsim_\mathfrak{t} t$.*
2. *$s \succ_\mathfrak{t} t$ if and only if $s' \succ_\mathfrak{t} t$.*
3. *$s \rhd_\mathfrak{t} t$ if and only if $s' \rhd_\mathfrak{t} t$.*

Proof. By mutual induction on the derivation for "if" and "only if" separately. Note that $\to_\kappa$ is confluent. □

The compatibility follows as a corollary:

Corollary 1. $\succsim_\mathfrak{t} ; \succ_\mathfrak{t} \subseteq \succ_\mathfrak{t} \cup (\succ_\mathfrak{t} ; \succ_\mathfrak{t})$.

Well-foundedness. Following [20], we base the well-foundedness proof of $\succ_\mathfrak{t}$ on the predicate of computability [34,16]. There are, however, two major differences, which pose new technical challenges: $\succsim_\mathfrak{t}$ is no more the reflexive closure of $\succ_\mathfrak{t}$ and curried notation instead of uncurried notation is in use.

In Definition 6, $\succ_\varphi^\mathfrak{f}$ and $\succ_\varphi^\mathfrak{m}$ are induced by $\succsim_\varphi$ and $\succ_\varphi$. We need certain properties of $\succ_\mathfrak{t}^\mathfrak{f}$ and $\succ_\mathfrak{t}^\mathfrak{m}$ to prove that $\succ_\mathfrak{t}$ is well-founded. Because $\succsim_\mathfrak{t}$ is neither the equality over terms nor the reflexive closure of $\succ_\mathfrak{t}$, those properties are less standard and deserve inspection. The property of $\succ_\mathfrak{t}^\mathfrak{f}$ is relatively easy to prove:

Theorem 5. *Given relations $\succsim$ and $\succ$ over X such that $\succ$ is well-founded and $\succsim ; \succ \subseteq \succ^+$, $\succ^\mathfrak{f}$ is well-founded over X^n for all n.*

Proof. The standard method used when $\succsim$ is the equality still applies. □

We refer to [35] for the proof of the following property of $\succ_\mathfrak{t}^\mathfrak{m}$:

Theorem 6. *Given relations $\succsim$ and $\succ$ over X such that $\succsim$ is a quasi-ordering, $\succ$ is well-founded and $\succsim ; \succ \subseteq \succ$, $\succ^\mathfrak{m}$ is well-founded over X^*.*

Proof. See Theorem 3.7 in [35]. □

In comparison to [35], we waive the transitivity requirement for $\succ$ above, but we cannot get around the requirement that $\succsim$ is a quasi-ordering without significantly changing the proof. This seems problematic because $\succsim_\mathfrak{t}$ is not necessarily transitive due to its inclusion of $\succ_\mathfrak{t}$. Fortunately, one observation resolves this issue: $\succ_\mathfrak{t}^\mathfrak{m}$ can equivalently be seen as induced by $\downarrow_\kappa$ and $\succ_\mathfrak{t}$ due to Theorem 4. In the same spirit, we can prove the following property:

Theorem 7. *$\downarrow_\kappa^\mathfrak{m} ; \succ_\mathfrak{t}^\mathfrak{m} \subseteq \succ_\mathfrak{t}^\mathfrak{m}$ where $s_1 \dots s_n \downarrow_\kappa^\mathfrak{m} t_1 \dots t_n$ if and only if there exists a permutation π over $\{1, \dots, n\}$ such that $s_{\pi(i)} \downarrow_\kappa t_i$ for all i.*

Proof. See Lemma 3.2 in [35]. □

Our definition of computability (or reducibility [16]) is standard:

Definition 7. *A term t_0 is called* computable *if either*

1. *the type of t_0 is a sort and t_0 is terminating with respect to $\succ_\mathfrak{t}$, or*
2. *the type of t_0 is $A \to B$ and $t_0\ t_1$ is computable for all computable $t_1 : A$.*

In [20], a term is called neutral if it is not a λ-abstraction. Due to the exclusion of λ-abstractions from LCSTRSs, one might consider all LCSTRS terms neutral. This naive definition, however, does not capture the essence of neutrality: if a term t_0 is neutral, a reduct (with respect to $\succ_\mathfrak{t}$) of $t_0\ t_1$ can only be $t_0'\ t_1'$ where t_0' and t_1' are reducts of t_0 and t_1, respectively. Because LCSTRSs use curried notation, neutral terms should be defined as follows:

Definition 8. *A term is called* neutral *if it assumes the form $x\ t_1 \cdots t_n$ for some variable x.*

And we recall the following results:

Theorem 8. *Computable terms have the following properties:*

1. *Given terms s and t such that $s \succ_\mathfrak{t} t$, if s is computable, so is t.*
2. *All computable terms are terminating with respect to $\succ_\mathfrak{t}$.*
3. *Given a neutral term s, if t is computable for all t such that $s \succ_\mathfrak{t} t$, so is s.*

Proof. The standard proof still works despite the seemingly different definition of neutrality. □

In addition, we prove the following lemma:

Lemma 5. *Given terms s and t such that $s \downarrow_\kappa t$, if s is computable, so is t.*

Proof. By induction on the type of s and t. □

And we have the following corollary due to Theorem 4:

Corollary 2. *Given terms s and t such that $s \succsim_\mathfrak{t} t$, if s is computable, so is t.*

The goal is to prove that all terms are computable. To do so, the key is to prove that $f\ s_1 \cdots s_m$ is computable where f is a function symbol if s_i is computable for all i. In [20], this is done on the basis that $f\ s_1 \cdots s_m$ is neutral, which is not true in our case. We do it differently and start with a definition:

Definition 9. *Given $f : A_1 \to \cdots \to A_n \to B$ where $f \in \mathcal{F}$ and $B \in \mathcal{S}$, let $\mathrm{ar}(f)$ be n. We introduce a special symbol $\top$ and extend our previous definitions so that $\top \succ_\mathfrak{t} t$ for all $t \in T(\mathcal{F}, \mathcal{V})$ and $\top \downarrow_\kappa \top$. This way $\top \succsim_\mathfrak{t} t$ if $t \in T(\mathcal{F}, \mathcal{V})$ or $t = \top$. Given terms $\bar{t} = t_1 \dots t_n$, let $(\bar{t})_k$ be t_k if $k \leq n$, and $\top$ if $k > n$. Given terms $s = f\ s_1 \cdots s_m$ and $t = g\ t_1 \cdots t_n$ where $f \in \mathcal{F}$, $g \in \mathcal{F}$, all s_i and t_i are computable, we define $\succ_c$ such that $s \succ_c t$ if and only if $f \blacktriangleright g$, or $f = g$ and*

- $\mathfrak{s}(f) = \mathfrak{l}$ *and* $(\bar{s})_1 \dots (\bar{s})_{\mathrm{ar}(f)} \succ_\mathfrak{t}^\mathfrak{l} (\bar{t})_1 \dots (\bar{t})_{\mathrm{ar}(f)}$, *or*
- $\mathfrak{s}(f) = \mathfrak{m}_k$ *and*
  - $(\bar{s})_1 \dots (\bar{s})_k \succ_\mathfrak{t}^\mathfrak{m} (\bar{t})_1 \dots (\bar{t})_k$, *or*
  - $(\bar{s})_1 \dots (\bar{s})_k \downarrow_\kappa^\mathfrak{m} (\bar{t})_1 \dots (\bar{t})_k$, $\forall i > k.\, (\bar{s})_i \succsim_\mathfrak{t} (\bar{t})_i$ *and* $\exists i > k.\, (\bar{s})_i \succ_\mathfrak{t} (\bar{t})_i$.

This gives us a well-founded relation:

Lemma 6. *$\succ_c$ is well-founded.*

Proof. Since all computable terms are terminating with respect to $\succ_\mathfrak{t}$, $\succ_\mathfrak{t}$ is well-founded over computable terms. The introduction of $\top$ clearly does not break this well-foundedness. The outermost layer of $\succ_c$ regards $\blacktriangleright$, which is well-founded by definition. We need only to fix the function symbol f and to go deeper. If $\mathfrak{s}(f) = \mathfrak{l}$, we know that $\succ_\mathfrak{t}^\mathfrak{l}$ is well-founded over lists of length $\mathrm{ar}(f)$ because of Theorem 5. If $\mathfrak{s}(f) = \mathfrak{m}_k$, $\succ_c$ splits each list of arguments in two and performs a lexicographic comparison. We can go past the first component because of Theorems 6 and 7. And the rest, a pointwise comparison, is also well-founded. So we can conclude that $\succ_c$ is well-founded. □

Now we prove the aforementioned statement:

Lemma 7. *Given a term $s = f\ s_1 \cdots s_m$ where f is a function symbol, if s_i is computable for all i, so is s.*

Proof. By well-founded induction on $\succ_c$. We consider the type of s:

- If the type is a sort, we ought to prove that s is terminating with respect to $\succ_\mathfrak{t}$. We need only to consider the cases in which s is not a theory term because all theory terms are terminating with respect to $\succ_\mathfrak{t}$ due to the well-foundedness of $[\![\sqsupset]\!]$. Take an arbitrary term t such that $s \succ_\mathfrak{t} t$. We prove that t is terminating with respect to $\succ_\mathfrak{t}$ by case analysis on the derivation of $s \succ_\mathfrak{t} t$. If $t = f\ t_1 \cdots t_m$, $\forall i.\, s_i \succsim_\mathfrak{t} t_i$ and $\exists k.\, s_k \succ_\mathfrak{t} t_k$, we can prove that $s \succ_c t$. By induction, t is computable and therefore terminating with respect to $\succ_\mathfrak{t}$. If $s \rhd_\mathfrak{t} t$, we prove that t is computable for all t such that $s \rhd_\mathfrak{t} t$ (t is generalized) by inner induction on the derivation of $s \rhd_\mathfrak{t} t$:
  1. If $\exists k.\, s_k \succsim_\mathfrak{t} t$, t is computable due to Corollary 2.
  2. If $t = t_0\ t_1 \cdots t_n$ and $\forall i.\, s \rhd_\mathfrak{t} t_i$, t_i is computable for all i by inner induction. By definition, t is computable.
  3. If $t = g\ t_1 \cdots t_n$, $f \blacktriangleright g$ and $\forall i.\, s \rhd_\mathfrak{t} t_i$, t_i is computable for all i by inner induction. It follows from $f \blacktriangleright g$ that $s \succ_c t$, and t is computable by outer induction.
  4. If $t = f\ t_1 \cdots t_n$, $\mathfrak{s}(f) = \mathfrak{l}$, $s_1 \dots s_m \succ_\mathfrak{t}^\mathfrak{l} t_1 \dots t_n$ and $\forall i.\, s \rhd_\mathfrak{t} t_i$, t_i is computable for all i by inner induction. Likewise, $s \succ_c t$.
  5. If $t = f\ t_1 \cdots t_n$, $\mathfrak{s}(f) = \mathfrak{m}_k$, $k \leq n$, $s_1 \dots s_{\min(m,k)} \succ_\mathfrak{t}^\mathfrak{m} t_1 \dots t_k$ and $\forall i.\, s \rhd_\mathfrak{t} t_i$, t_i is computable for all i by inner induction. Likewise, $s \succ_c t$.
  6. If t is a value, t is terminating with respect to $\succ_\mathfrak{t}$ and its type is a sort.
- If the type is $A \to B$, take an arbitrary computable $s_{m+1} : A$. We prove that $s \succ_c s\ s_{m+1} = f\ s_1 \cdots s_{m+1}$. Note that $(s_1 \dots s_m)_i = (s_1 \dots s_{m+1})_i$ for all $i \leq m$ and $(s_1 \dots s_m)_{m+1} = \top \succ_\mathfrak{t} (s_1 \dots s_{m+1})_{m+1}$. Consider $\mathfrak{s}(f) = \mathfrak{l}$, $\mathfrak{s}(f) = \mathfrak{m}_k$ while $k > m$, and $\mathfrak{s}(f) = \mathfrak{m}_k$ while $k \leq m$. We have $s \succ_c s\ s_{m+1}$ in each case. By induction, $s\ s_{m+1}$ is computable. Hence, s is computable.

We conclude that s is computable. □

Now the well-foundedness of $\succ_\mathfrak{t}$ follows immediately:

Theorem 9. *$\succ_\mathfrak{t}$ is well-founded.*

Proof. We prove that every term t is computable by induction on t. Given Lemma 7, we need only to prove that variables are computable, which is the case because variables are neutral and in normal form with respect to $\succ_\mathfrak{t}$. □

5 Implementation

A preliminary implementation of LCSTRSs is available in Cora, at:

(URL censored; an anonymised copy of the repository is available as supplementary material.)

Thus far, Cora (*Constrained Rewriting Analyser*) supports only integers and booleans, but is intended to eventually support arbitrary theories provided an SMT solver is available. Example input files are supplied in the repository. Cora is open-source, and freely available as a library or a stand-alone tool—with the caution that the core libraries are still in active development, and liable to change in future versions. Nevertheless, Cora is already used in several student projects.

Automating constrained HORPO Cora includes an implementation of constrained HORPO. Following the example of existing termination tools such as AProVE [14], Natt [36] and Wanda [23], we use an encoding to SMT, so that a satisfying assignment to the variables in the SMT problem delivers a precedence, status, and the relation $\sqsupseteq_{\mathsf{int}}$ for a successful constrained HORPO proof. (We simply fix the boolean ordering as $1 \sqsupseteq_{\mathsf{bool}} 0$.)

To encode the precedence and status, we introduce integer variables $\mathtt{pred}_f$ and $\mathtt{status}_f$ for all symbols other than values, and we let $\mathtt{down}$ be a boolean variable that indicates two possible choices for $\sqsupseteq_{\mathsf{int}}$ (counting up or down). We require that $\mathtt{pred}_f < 0$ for theory symbols and $\mathtt{pred}_f \geq 0$ otherwise, and that $1 \leq \mathtt{status}_f \leq m$, where m is the maximum number of arguments f can take. Here, $\mathtt{status}_f = 1$ indicates $\mathfrak{l}$ status, and $\mathtt{status}_f = i > 1$ indicates status $\mathfrak{m}_i$.

Then, we make the following observation: if $uR_\psi v$ occurs anywhere in the proof of $s \succ_\varphi t$, then $\psi = \varphi$ and u, v are subterms of s, t. Hence, for a given LC-STRS there are only finitely many possible inequalities to analyze. By inspecting the definition of constrained HORPO, we also note that there are no cyclic dependencies. Hence, we can enumerate all pairs $u\ R_\varphi\ v$ where $\ell \to r\ [\varphi] \in \mathcal{R}$, u is a subterm of s, v a subterm of t, and $R \in \{\succsim, \succ, \triangleright, 1a, 1b, 1c, 1d, 2a, 2b, 2c, 2d, 3a, 3b, 3d, 3e, 3f\}$. (Although in practice, we do not blindly enumerate them but only generate the ones that may actually appear in a HORPO proof.) We introduce a variable $\langle u\ R_\varphi\ v\rangle$ for each, and add defining constraints. We will not go in detail for all cases, but provide some key examples:

- If u and v do not have the same type structure, then we add $\neg\langle u \succsim_\varphi v\rangle$; otherwise we add $\langle u \succsim_\varphi v\rangle \Rightarrow \langle u\ 1a_\varphi\ u\rangle \vee \langle u\ 1b_\varphi\ b\rangle \vee \langle u\ 1c_\varphi\ v\rangle \vee \langle u\ 1d_\varphi\ v\rangle$. This states that for $u \succsim_\varphi v$ to hold, it must hold by one of the cases 1a, 1b, 1c, 1d. Each of these cases also has a defining constraint.
- $\langle f\ \vec{u}\ 3c_\varphi\ g\ v_1 \cdots v_m\rangle \Rightarrow \mathtt{pred}_f > \mathtt{pred}_g \wedge \langle f\ \vec{u} \triangleright_\varphi v_1\rangle \wedge \cdots \wedge \langle f\ \vec{u} \triangleright_\varphi v_m\rangle$
  This states for $f\ u_1 \cdots u_n \triangleright_\varphi g\ v_1 \cdots v_m$ to hold by rule 3c, we require $f \blacktriangleright g$ and $f\ u_1 \cdots u_n \triangleright_\varphi v_i$ for $1 \leq i \leq m$.
- To give a defining constraint for $\langle u 2a_\varphi v\rangle$, we consider u, v and φ:
  - If u and v are not both theory terms of the same theory sort, or if $\mathrm{Var}(u) \cup \mathrm{Var}(v) \not\subseteq \mathrm{Var}(\varphi)$, we add the constraint: $\neg\langle u\ 2a_\varphi\ v\rangle$.
  - Otherwise, if they have type int we check if $\varphi \Rightarrow u \geq -M \wedge u > v$ is valid, and if not: we add the constraint $\langle u\ 2a_\varphi\ v\rangle \Rightarrow \neg\mathtt{down}$.
    We also check if $\varphi \Rightarrow u \leq M \wedge u < v$ is valid, and if not, add the constraint $\langle u\ 2a_\varphi\ v\rangle \Rightarrow \mathtt{down}$.

- If they have type bool, we check if $\varphi \Rightarrow u \wedge \neg v$ is valid, and if not, we add the constraint $\neg\langle u\ 2a_\varphi v\rangle$. (If it is valid, we add nothing, allowing the SMT solver to set the variable $\langle u\ 2a_\varphi v\rangle$ to true.)

Here, the bound M is twice the largest absolute integer value occurring in the rules, or 1000 if that is larger. (This value is chosen arbitrarily.) Note that we do *not* include the validity checks directly in the SMT problem: this would require a universal quantification within a satisfiability proof, which is typically hard for SMT solvers. Rather, we pose a separate question to the solver for each theory comparison we may encounter, and consider whether the pair can be oriented downwards (using $\sqsupset_{\mathsf{int}} := \{(x,y) \mid x > -M \wedge x > y\}$), upwards (using $\sqsupset_{\mathsf{int}} := \{(x,y) \mid x < M \wedge x < y\}$), or not at all. This is why we must fix the bound M beforehand.

– The hardest case is 3e: a multiset comparison, where we not only need to implement a multiset comparison as an SMT problem, but also take into account that for a status $\mathfrak{m}_k$ only k arguments on each side should be considered. To give a defining constraint for $\langle f\ u_1 \cdots u_m\ 3e_\varphi\ f\ v_1 \cdots v_n\rangle$, we introduce new boolean variables $\mathtt{strict}_1, \ldots, \mathtt{strict}_m$ (where $\mathtt{strict}_i$ indicates that $i \in I$ following Definition 4) and integer variables $\pi(1), \ldots, \pi(n)$ and require:
  - $\langle f\ \vec{u}\ 3e_\varphi\ f\ \vec{v}\rangle \Rightarrow \mathtt{status}_f \geq 2$
  - $\langle f\ \vec{u}\ 3e_\varphi\ f\ \vec{v}\rangle \Rightarrow \mathtt{status}_f \leq n$
  - $\langle f\ \vec{u}\ 3e_\varphi f\ \vec{v}\rangle \Rightarrow \langle f\ \vec{u} \rhd_\varphi v_1\rangle \wedge \cdots \wedge \langle f\ \vec{u} \rhd_\varphi\ v_n\rangle$
  - $\langle f\ \vec{u}\ 3e_\varphi\ f\ \vec{v}\rangle \Rightarrow \mathtt{strict}_1 \vee \cdots \vee \mathtt{strict}_m$ and $\langle f\ \vec{u}\ 3e_\varphi\ f\ \vec{v}\rangle \wedge \mathtt{strict}_j \Rightarrow \mathtt{status}_f \geq j$ for all $j \in \{1, \ldots, m\}$ (that is, if f has status $\mathfrak{m}_k$, then $I \subseteq \{1, \ldots, k\}$)
  - for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$:
    * $\langle f\ \vec{u}\ 3e_\varphi\ f\ \vec{v}\rangle \wedge i \leq \mathtt{status}_f \Rightarrow 1 \leq \pi(f) \wedge \pi(f) \leq \mathtt{status}_f \wedge \pi(f) \leq m$ (that is, if f has status $\mathfrak{m}_k$, then $\pi(i) \in \{1, \ldots, \min(m,k)\}$)
    * $\langle f\ \vec{u}\ 3e_\varphi\ f\ \vec{v}\rangle \Rightarrow \pi(i) \neq j \vee \pi(i') \neq j \vee \neg\mathtt{strict}_j$ for all $i_1 < i_2 \leq n$ (that is, for all $j \in \{1, \ldots, m\} \setminus I$: $|\pi^{-1}(j)| \leq 1$—this suffices because if $|\pi^{-1}(j)| = 0$ we can include j in I without changing the relation).
    * $\langle f\ \vec{u}\ 3e_\varphi\ f\ \vec{v}\rangle \wedge \pi(i) = j \wedge \mathtt{strict}_j \Rightarrow \langle u_j \succ v_i\rangle$ (for $j \in I$ and $i \in \pi^{-1}(j)$: $u_j \succ v_i$)
    * $\langle f\ \vec{u}\ 3e_\varphi\ f\ \vec{v}\rangle \wedge \pi(i) = j \wedge \neg\mathtt{strict}_j \Rightarrow \langle u_j \succsim v_i\rangle$ (for $j \in \{1, \ldots, m\} \setminus I$ and $i \in \pi^{-1}(j)$: $u_j \succsim v_i$)

Cora succeeds in proving termination for all examples in this paper except Example 2 (which is non-terminating). The input files are included in the repository.

6 Related Work

In this section, we assess the newly proposed formalism and the prospects for its application by comparing and relating it to the literature.

Term rewriting. The closest related work are LCTRSs [12], a first-order formalism for constrained rewriting that the present work builds upon. Similarly, there are many formalisms of higher-order term rewriting, but without in-built constraints (e.g., [20,21,26]). It seems likely that the analysis methods for these can be expanded with support for SMT, as was done for HORPO in this paper.

Also worth mentioning is the K Framework [29] which, like LCSTRSs, can be used as an intermediate language for program analysis and is based on a form of first-order rewriting. The K tool includes analysis techniques through *reachability logic*, rather than methods like HORPO.

There are several works that analyze properties of (functional and other) programs using different forms of term rewriting (e.g., [2,15]). However, these works typically consider translations to first-order rewriting; hence, some of the structure of the initial problem is lost, and analysis power is sacrificed.

HORPO. Our definition of constrained HORPO builds on the first-order constrained RPO for LCTRSs in [24] and the first definition of higher-order RPO [20]. However, there have been other HORPO extensions since, e.g., [4,5]. We believe that the ideas for these extensions can also be used for LCSTRSs. We have not done so because the purpose of this paper is to show *that* and *how* analysis techniques for higher-order rewriting extend; not to introduce the most powerful (and consequently more elaborate) ones.

Also worth mentioning is [3], a higher-order RPO for applicative systems. This variant is defined for superposition rather than termination analysis, and is ground-total, but at the price of not being stable, making it less suitable.

Functional programming. There are many works on analyzing functional programs directly, including termination, although this typically considers *specific* programming languages such as OCaml (e.g., [19]) or Haskel (e.g., [18]). A variety of techniques are used, such as sized types [27] or decreasing measures on data [17], but for what we could find, there is no real parallel of many rewriting techniques such as RPO. Our hope is that, through LCSTRSs, we can help to make the methods of term rewriting available to this community.

7 Conclusion and Future Work

In summary, we have defined a higher-order extension of logically constrained term rewriting systems, which can represent realistic higher-order programs in a natural way. To illustrate how such systems may be analyzed, we have adapted HORPO, one of the oldest higher-order termination techniques, to handle constraints. Despite being a very basic method, this is already powerful enough to handle most examples in this paper. Both LCSTRSs and constrained HORPO are implemented in our new analysis tool Cora.

In the future, we intend to extend other analysis techniques for both first-order and higher-order term rewriting to this formalism, and implement them

into a fully automatic tool. We hope that this will make the methods of the term rewriting community available to other communities; both by providing a powerful backend tool, and by showing how existing techniques can be adapted so they may also be natively adopted in analyses of functional languages.

A natural starting point is to increase our power in termination analysis by extending *dependency pairs* [1,33,11,22] and various supporting methods like the subterm criterion and usable rules. In addition, methods for complexity, reachability and equivalence (e.g., through *rewriting induction* [28,12]) which have been defined for first-order LCTRSs, are natural directions for higher-order extensions as well.

References

1. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science **236**(1-2), 133–178 (2000). https://doi.org/10.1016/S0304-3975(99)00207-8
2. Avanzini, M., Dal Lago, U., Moser, G.: Analysing the complexity of functional programs: Higher-order meets first-order. In: Proc. ICFP. pp. 152–164. ACM (2015). https://doi.org/10.1145/2784731.2784753
3. Blanchette, J., Waldmann, U., Wand, D.: A lambda-free higher-order recursive path order. In: Esparza, J., Murawski, A. (eds.) Proc. FoSSaCS. LNTCS, vol. 10203, pp. 461–479 (2017). https://doi.org/10.1007/978-3-662-54458-7_27
4. Blanqui, F., Jouannaud, J., Rubio, A.: HORPO with computability closure: A reconstruction. In: Dershowitz, N., Voronkov, A. (eds.) Proc. LPAR. LNAI, vol. 4790, pp. 138–150 (2007). https://doi.org/10.1007/978-3-540-75560-9_12
5. Blanqui, F., Jouannaud, J., Rubio, A.: The computability path ordering: The end of a quest. In: Kaminski, M., Martini, S. (eds.) Proc. CSL. LNCS, vol. 5213, pp. 1–14 (2008). https://doi.org/10.1007/978-3-540-87531-4_1
6. Ciobâcă, S., Lucano, D., Buruiană, A.: Operationally-based program equivalence proofs using LCTRSs. Journal of Logical and Algebraic Methods in Programming (JLAMP) **135** (2023). https://doi.org/10.1016/j.jlamp.2023.100894
7. Community: Smt-lib, http://www.smtlib.org/
8. Falke, S., Kapur, D.: A term rewriting approach to the automated termination analysis of imperative programs. In: Schmidt, R.A. (ed.) Proc. CADE. pp. 277–293 (2009). https://doi.org/10.1007/978-3-642-02959-2_22
9. Falke, S., Kapur, D.: Rewriting induction + linear arithmetic = decision procedure. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Proc. IJCAR. pp. 241–255 (2012). https://doi.org/10.1007/978-3-642-31365-3_20
10. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: Schmidt-Schauß, M. (ed.) Proc. RTA. pp. 41–50 (2011). https://doi.org/10.4230/LIPIcs.RTA.2011.41
11. Fuhs, C., Kop, C.: A static higher-order dependency pair framework. In: Caires, L. (ed.) Proc. ESOP. LNTCS, vol. 11423, pp. 752–782 (2019). https://doi.org/10.1007/978-3-030-17184-1_27
12. Fuhs, C., Kop, C., Nishida, N.: Verifying procedural programs via constrained rewriting induction. ACM TOCL **18**(2), 14:1–14:50 (2017). https://doi.org/10.1145/3060143

13. Furuichi, Y., Nishida, N., Sakai, M., Kusakari, K., Sakabe, T.: Approach to procedural-program verification based on implicit induction of constrained term rewriting systems. IPSJ Trans. Program. **1**(2), 100–121 (2008), in Japanese
14. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity automatically with AProVE. Journal of Automated Reasoning (JAR) **58**, 3–31 (2017). https://doi.org/10.1007/s10817-016-9388-y
15. Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., Thiemann, R.: Automated termination proofs for Haskell by term rewriting. ACM Transactions on Programming Languages and Systems (TOPLAS) **33**(2), 7:1–7:39 (2011). https://doi.org/10.1145/1890028.1890030
16. Girard, J.Y., Taylor, P., Lafont, Y.: Proofs and Types. Cambridge University Press (1989)
17. Hamza, J., Voirol, N., Kunčak, V.: System fr: formalized foundations for the stainless verifier. ACM OOPSLA **3**, 166:1–166:30 (2019). https://doi.org/10.1145/3360592
18. Handley, M.A.T., Vazou, N., Hutton, G.: Liquidate your assets: Reasoning about resource usage in liquid haskell. ACM POPL **4**, 24:1–24:27 (2019). https://doi.org/10.1145/3371092
19. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ml. In: Madhusudan, P., Seshia, S. (eds.) Proc. CAV. LNTCS, vol. 7358, pp. 781–786 (2012). https://doi.org/10.1007/978-3-642-31424-7_64
20. Jouannaud, J.P., Rubio, A.: The higher-order recursive path ordering. In: Longo, G. (ed.) Proc. LICS. pp. 402–411 (1999). https://doi.org/10.1109/LICS.1999.782635
21. Klop, J., Oostrom, V.v., Raamsdonk, F.v.: Combinatory reduction systems: introduction and survey. Theoretical Computer Science **121**(1-2), 279 – 308 (1993). https://doi.org/10.1016/0304-3975(93)90091-7
22. Kop, C.: Termination of LCTRSs. In: Waldmann, J. (ed.) Proc. WST. pp. 59–63 (2013). https://doi.org/10.48550/arXiv.1601.03206
23. Kop, C.: WANDA – a higher order termination tool. In: Ariola, Z. (ed.) Proc. FSCD. LIPIcs, vol. 167, pp. 36:1–36:19 (2020). https://doi.org/10.4230/LIPIcs.FSCD.2020.36
24. Kop, C., Nishida, N.: Term rewriting with logical constraints. In: Fontaine, P., Ringeissen, C., Schmidt, R.A. (eds.) Proc. FroCoS. pp. 343–358 (2013). https://doi.org/10.1007/978-3-642-40885-4_24
25. Kusakari, K.: On proving termination of term rewriting systems with higher-order variables. IPSJ Transactions on Programming **42**(SIG 7 PRO11), 35–45 (2001), http://id.nii.ac.jp/1001/00016864/
26. Nipkow, T.: Higher-order critical pairs. In: Kahn, G. (ed.) Proc. LICS. pp. 342–349. IEEE (1991). https://doi.org/10.48456/tr-218
27. Pareto, L.: Sized types (1998), dissertation for the Licentiate Degree in Computing Science, Chalmers University of Technology
28. Reddy, U.S.: Term rewriting induction. In: Stickel, M. (ed.) Proc. CADE. LNCS, vol. 449, pp. 162–177 (1990). https://doi.org/10.1007/3-540-52885-7_86
29. Rosu, G., Florin Șerbănuță, T.: An overview of the K semantic framework. The Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
30. Sakata, T., Nishida, N., Sakabe, T., Sakai, M., Kusakari, K.: Rewriting induction for constrained term rewriting systems. IPSJ Trans. Program. **2**(2), 80–96 (2009), in Japanese

31. Schneider-Kamp, P., Giesl, J., Ströder, T., Serebrenik, A., Thiemann, R.: Automated termination analysis for logic programs with cut. TPLP **10**(4–6), 365–381 (2010). https://doi.org/10.1017/S1471068410000165
32. Schöpf, J., Middeldorp, A.: Confluence criteria for logically constrained rewrite systems. In: Proc. CADE. LNAI, vol. 14132, pp. 474–490 (2023). https://doi.org/10.1007/978-3-031-38499-8_27
33. Suzuki, S., Kusakari, K., Blanqui, F.: Argument filterings and usable rules in higher-order rewrite systems. IPSJ Transactions on Programming **4**(2), 1–12 (2011). https://doi.org/10.2197/ipsjtrans.4.114
34. Tait, W.W.: Intensional interpretations of functionals of finite type I. JSL **32**(2), 198–212 (1967). https://doi.org/10.2307/2271658
35. Thiemann, R., Allais, G., Nagele, J.: On the formalization of termination techniques based on multiset orderings. In: Tiwari, A. (ed.) Proc. RTA. pp. 339–354 (2012). https://doi.org/10.4230/LIPIcs.RTA.2012.339
36. Yamada, A., Kusakari, K., Sakabe, T.: Nagoya termination tool. In: Proc. RTA-TLCA. LNTCS, vol. 8560, pp. 466–475 (2014). https://doi.org/10.1007/978-3-319-08918-8_32