Higher-Order LCTRSs and Their Termination

Liye Guo[0000−0002−3064−2691] and Cynthia Kop[0000−0002−6337−2544]

Radboud University, Nijmegen, The Netherlands
{l.guo,c.kop}@cs.ru.nl

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 example, termination (see, e.g., [8,10,15,37]) and equivalence (see, e.g., [13,36,9]). 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; think of $\mathsf{neg}\ (\mathsf{suc}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero})))$ encoding -3. 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 transfer techniques between such formalisms.

Logically constrained term rewriting systems (LCTRSs) [27,12] 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 (see, e.g., [32,24,23]). In particular, rewriting induction on LCTRSs [12,30] offers a powerful tool for program verification.

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

Motivation. Below is a first-order LCTRS implementing the factorial function:

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

where $n \le 0$ and $n > 0$ are logical constraints, which the integer n must satisfy respectively when the corresponding rewrite rule is applied. Suppose we have access to higher-order functions, a defining feature of functional programming; now we have the following alternative implementation of fact:

$$\mathsf{fact}\ n \to \mathsf{fold}\ (*)\ 1\ (\mathsf{genlist}\ n)$$
$$\mathsf{genlist}\ n \to \mathsf{nil} \quad [n \le 0] \qquad \mathsf{genlist}\ n \to \mathsf{cons}\ n\ (\mathsf{genlist}\ (n-1)) \quad [n > 0]$$
$$\mathsf{fold}\ f\ y\ \mathsf{nil} \to y \qquad \mathsf{fold}\ f\ y\ (\mathsf{cons}\ x\ l) \to f\ x\ (\mathsf{fold}\ f\ y\ l)$$

Here fold takes an argument f, which itself represents a function. Higher-order functions such as fold do not fit into first-order LCTRSs, which leads to the first reason to generalize this formalism: to overcome the limitation of its expressivity.

There is another reason for higher-order LCTRSs. The latter implementation of fact reflects a pattern of functional programming: the combination of "standard" higher-order building blocks such as fold and map, and functions that are specific to the problem at hand. We note that a higher-order formalism can reveal more modularity in programs. It would be valuable to exploit such modularity in *analyses* as well.

With higher-order LCTRSs, we would like to explore automatable solutions to the termination problem of functional programs in the same fashion as the first-order case [27,25], or even better, to the finding of their complexity by term rewriting. Moreover, given two programs supposedly implementing the same function, a method that derives whether they are indeed equivalent is also desirable. For example, a proof that the above two implementations of fact are equivalent may serve as a correctness proof of the latter, less intuitive implementation (which in general might be an outcome of code refactoring). Such methods have been explored in a first-order setting [12,7].

Higher-order LCTRSs will broaden the horizons of both LCTRSs and higher-order term rewriting. 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. The presentation begins with our perspective on higher-order term rewriting (without logical constraints) in Section 2. The contributions of this paper follow in subsequent sections:

– We propose the formalism of *logically constrained simply-typed term rewriting systems* (LCSTRSs), a higher-order generalization of LCTRSs, in Section 3.
– We adapt *reduction orderings* and *rule removal* to the newly proposed formalism, and define (as well as prove the soundness of) *constrained HORPO*—a variant of HORPO [21]—in Section 4. This includes changes to fit HORPO to curried notation and partial application, and to handle theory symbols and logical constraints in a similar way to RPO for first-order LCTRSs [27]. While this version of HORPO is not the most powerful higher-order termination technique, it offers a simple yet self-contained solution, and serves to illustrate how existing techniques may be extended.
– We have developed for our formalism the foundation of a new open-source analysis tool, in which an implementation of constrained HORPO is provided. It requires several new insights, especially with regard to the way theories and logical constraints are handled, and is discussed in Section 6.

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 (STRSs) [29] as the unconstrained basis of our formalism. This is one of the simplest higher-order formalisms, and closely resembles simple functional programs. We choose this formalism as our starting point because it is already powerful, while avoiding many of the complications that may be interesting for equational reasoning purposes but are not needed in 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 $t_0\ t_1\ t_2$ stands for $((t_0\ t_1)\ t_2)$, for example.

We assume that every function symbol and variable is assigned a unique type. Typing 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 $T(\mathcal{F}, \emptyset)$.

Substitutions and Contexts. Variables occurring in a term can be seen as placeholders: the occurrences of a variable 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 $\square$ be a special terminal symbol denoting the hole, and the grammar $\mathfrak{C} ::= \square \mid (\mathfrak{C}\ \mathfrak{T}) \mid (\mathfrak{T}\ \mathfrak{C})$ 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: $\square[t]_A = t$, $(C_0[]_A\ t_1)[t]_A = C_0[t]_A\ t_1$ 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 Rewriting. 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 system 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 system 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 system:

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

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 system:

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

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 above formalism 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 higher-order formalisms in future extensions.

The exclusion of λ-abstractions does not rid us of first-class functions, thanks to curried notation. For example, the occurrence of suc in $\mathsf{iterate\ suc\ zero}$ 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 of encoding product types.

Notions of Termination. If we combine the two systems from Examples 1 and 2, the outcome is surely non-terminating: $\mathsf{take\ zero}\ (\mathsf{iterate\ suc\ zero})$ 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 primitive data types built in; with some function symbols *constructing* (introducing) values of a certain type and pattern matching rules *deconstructing* (eliminating) those values, a term rewriting system relies on (co)inductively defined data types. While (co)inductive reasoning is straightforward this way, data types such as integers and arrays require encoding, which can be convoluted; think of the space-consuming unary representation of a number or a binary representation which takes less space but shifts the burden to rewrite rules defining arithmetic, and negative numbers bring up even more complications. Besides, such encoding neglects advances in modern SMT solvers.

In this section, we extend unconstrained STRSs with logical constraints so that data types that are not (co)inductively defined can be represented directly, and analysis tools can take advantage of existing SMT solvers. We follow the ideas of first-order LCTRSs [27,12]. Specifically, we will consider systems over *arbitrary* first-order theories, i.e., we are not bound to, say, systems over integers, while avoiding higher-order logical constraints. In the unconstrained part of such a system (outside theories), however, higher-order arguments and results are still completely usable.

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 $\mathcal{S}_\vartheta$. Note that the order of a theory type is never greater than one. 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 of type A an element of $\mathfrak{X}_A$ and is bijective[1] 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 integer arrays, we could either represent them as lists and simulate random access through more costly list traversal (which affects the complexity), or consider a theory of bounded arrays as follows: Let $\mathcal{S}_\vartheta$ be $\{\,\mathsf{int}, \mathsf{intarray}\,\}$ and $\mathcal{F}_\vartheta$ be 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 $\llbracket\cdot\rrbracket_{\mathsf{intarray}}$ be 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 $0 \leq i < k$, and 0 otherwise. Let $\llbracket \mathsf{store} \rrbracket(n_0 \ldots n_{k-1}, i, m)$ be $n_0 \ldots n_{i-1} m n_{i+1} \ldots n_{k-1}$ if $0 \leq i < k$, and $n_0 \ldots n_{k-1}$ otherwise. Note that the values include *theory symbols* $\langle \bar{n}_0, \ldots, \bar{n}_{k-1} \rangle : \mathsf{intarray}$ as well as $\bar{n} : \mathsf{int}$.

[1] The bijectivity is assumed so that values (see below) are isomorphic to (and therefore a representation of) elements of $(\mathfrak{X}_A)_{A \in \mathcal{S}_\vartheta}$.

In this paper, we largely consider the theory of integers in Example 3 when giving examples because it is easy to understand. This particular theory does not play a special role for the formalism we will shortly present; in fact, the theory of *bit vectors* may be more appropriate to real-world programs using integers, and our formalism is not biased toward any choice of theories. In particular, we do not have to choose predefined theories from SMT-LIB [3]. The theory of bounded arrays in Example 4 is an instance of such a "non-standard" theory (which can nevertheless be encoded within the theory of functional arrays). On the other hand, theories supported by SMT solvers are preferable in light of automation.

3.2 Constrained Rewriting

Constrained rewriting requires the theory sort bool: we henceforth assume that $\mathsf{bool} \in \mathcal{S}_\vartheta$, $\{\, \mathfrak{f}, \mathfrak{t} \,\} \subseteq \mathcal{F}_\vartheta$, $\mathfrak{X}_{\mathsf{bool}} = \{\, 0, 1 \,\}$, $[\![\mathfrak{f}]\!]_{\mathsf{bool}} = 0$ and $[\![\mathfrak{t}]\!]_{\mathsf{bool}} = 1$. 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 a term that assumes the form $f\ t_1 \cdots t_n$ for some function symbol f and contains at least one function symbol in $\mathcal{F} \setminus \mathcal{F}_\vartheta$.[2]

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,*

[2] We do not require f to be in $\mathcal{F} \setminus \mathcal{F}_\vartheta$ (that is, f can be a theory symbol) because a theory symbol may occur at the head position of a rewrite rule's left-hand side in rewriting induction, and this general definition is in line with first-order LCTRSs.

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 system.

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 *$\rightarrow_{\mathcal{R}} \subseteq T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V})$ such that $t \rightarrow_{\mathcal{R}} t'$ if and only if one of the following conditions is true:*

1. *There exist a rewrite rule $\ell \rightarrow r\ [\varphi] \in \mathcal{R}$, a substitution σ which respects $\ell \rightarrow r\ [\varphi]$ 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 $f : A_1 \rightarrow \cdots \rightarrow A_n \rightarrow B$ 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 are mutually exclusive for any given context $C[]$: $f\ v_1 \cdots v_n$ is a theory term, whereas ℓ in any rewrite rule $\ell \rightarrow r\ [\varphi]$ is not. If $t \rightarrow_{\mathcal{R}} t'$ due to the second condition, we also write $t \rightarrow_\kappa t'$ and call it a step of calculation. *When no ambiguity arises, we may simply write $\rightarrow$ for $\rightarrow_{\mathcal{R}}$.*

Example 5. We can rework Example 1 into an LCSTRS:

$$\begin{array}{rll} \mathsf{take}\ n\ l \rightarrow \mathsf{nil} & [n \le 0] & \mathsf{take}\ n\ \mathsf{nil} \rightarrow \mathsf{nil} \\ \mathsf{take}\ n\ (\mathsf{cons}\ x\ l) \rightarrow \mathsf{cons}\ x\ (\mathsf{take}\ (n-1)\ l) & [n > 0] & \end{array}$$

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

Here and henceforth we let integer literals and operators, e.g., 0, 1, $\le$, $>$ 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:

$$\begin{array}{rl} & \mathsf{take}\ 1\ (\mathsf{cons}\ x\ (\mathsf{cons}\ y\ l)) \rightarrow \mathsf{cons}\ x\ (\mathsf{take}\ (1-1)\ (\mathsf{cons}\ y\ l)) \\ \rightarrow_\kappa & \mathsf{cons}\ x\ (\mathsf{take}\ 0\ (\mathsf{cons}\ y\ l)) \rightarrow \mathsf{cons}\ x\ \mathsf{nil} \end{array}$$

Example 6. In Section 1, the rewrite rules implementing the factorial function by fold constitute an LCSTRS. Below is a rewrite sequence:

$$\begin{array}{l} \mathsf{fact}\ 1 \rightarrow \mathsf{fold}\ (*)\ 1\ (\mathsf{genlist}\ 1) \rightarrow \mathsf{fold}\ (*)\ 1\ (\mathsf{cons}\ 1\ (\mathsf{genlist}\ (1-1))) \\ \quad \rightarrow_\kappa \mathsf{fold}\ (*)\ 1\ (\mathsf{cons}\ 1\ (\mathsf{genlist}\ 0)) \rightarrow \mathsf{fold}\ (*)\ 1\ (\mathsf{cons}\ 1\ \mathsf{nil}) \\ \quad \rightarrow (*)\ 1\ (\mathsf{fold}\ (*)\ 1\ \mathsf{nil}) \rightarrow (*)\ 1\ 1 \rightarrow_\kappa 1 \end{array}$$

Example 7. Consider the rewrite rule $\mathsf{readint} \to n$, in which the variable $n : \mathsf{int}$ occurs on the right-hand side of → but not on the left. Unconstrained STRSs do not permit such a rewrite rule, but LCSTRSs do. It looks as if we might rewrite $\mathsf{readint}$ to a variable but it 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 non-deterministic. Even in the presence of an evaluation strategy, the order may not be the desired one. We can use continuation-passing style to choose an order:

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

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 demonstrates that the newly proposed formalism 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 LCTRSs can be seen as instances of the newly proposed formalism, i.e., ones in which 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.
- 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:

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

  The filter function can actually be implemented in an LCSTRS as follows:

  $$\begin{gathered}\mathsf{filter}\ f\ (\mathsf{cons}\ x\ l) \to \mathsf{if}\ (f\ x)\ (\mathsf{cons}\ x\ (\mathsf{filter}\ f\ l))\ (\mathsf{filter}\ f\ l)\\ \mathsf{filter}\ f\ \mathsf{nil} \to \mathsf{nil} \qquad \mathsf{if}\ \mathfrak{t}\ l\ l' \to l \qquad \mathsf{if}\ \mathfrak{f}\ l\ l' \to l'\end{gathered}$$

In the former implementation, 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 Constrained Higher-Order Recursive Path Ordering

Recall that an important part of our goal is to allow the abundant term rewriting techniques to be applied toward program analysis. We have defined a formalism for constrained higher-order term rewriting; now it remains to be seen that—or *how*—existing techniques can be extended to it.

In the rest of this paper, we consider termination, an important aspect of program analysis and a topic that 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 adapt HORPO [21] to our formalism. This is one of the oldest, yet still effective techniques for higher-order termination. HORPO can be used either as a stand-alone method or in a higher-order version of the dependency pair framework [1,39,11,25]. Hence, this adaptation offers a solid basis for use in an analysis tool's termination module. We will discuss the use of HORPO within the dependency pair framework in Section 5, and its automation in Section 6.

Constrained RPO for first-order LCTRSs was proposed in [27]. We take inspiration from it for its approach to theory terms, formalize the ideas, and add support for (higher) types as well as 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. $\forall i \in \{ 1, \ldots, m \} \setminus I. \forall j \in \pi^{-1}(i). \, x_i \succsim y_j$, *and*
3. $\forall i \in \{ 1, \ldots, m \} \setminus I. \, \left| \pi^{-1}(i) \right| = 1$.

Here the generalized multiset ordering is formulated in terms of lists because we will compare argument lists by this ordering and this formulation facilitates implementation. In the following definition of HORPO, when we refer to the above definitions, $\succsim$ is the *equality* over terms and $\succ$ is HORPO itself.

Roughly, HORPO extends a given ordering over function symbols, and when considering terms headed by the same function symbol, compares the arguments by either of the above orderings. 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. $s = f(s_1, \ldots, s_m)$, $f \in \mathcal{F}$ and $\exists k. \, s_k \succeq t$.
2. $s = f(s_1, \ldots, s_m)$, $f \in \mathcal{F}$, $t = @(\ldots @(@(t_0, t_1), t_2) \ldots, t_n)$ and $\forall i. \, s \succ t_i \vee \exists k. \, s_k \succeq t_i$.
3. $s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, $f \in \mathcal{F}$, $g \in \mathcal{F}$, $f \blacktriangleright g$ and $\forall i. \, s \succ t_i \vee \exists k. \, s_k \succeq t_i$.
4. $s = f(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_m)$, $f \in \mathcal{F}$, $\mathfrak{s}(f) = \mathfrak{l}$, $s_1 \ldots s_m \succ^{\mathfrak{l}} t_1 \ldots t_m$ and $\forall i. \, s \succ t_i \vee \exists k. \, s_k \succeq t_i$.
5. $s = f(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_m)$, $f \in \mathcal{F}$, $\mathfrak{s}(f) = \mathfrak{m}$ and $s_1 \ldots s_m \succ^{\mathfrak{m}} t_1 \ldots t_m$.
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 $f(s_1, \ldots, s_m)$ as opposed to $f \; s_1 \cdots s_m$. If f has arity m, its occurrence in a term must take m arguments so $f(s_1, \ldots, s_{m-1})$ is not a well-formed term, for example. 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 our extension of HORPO. Another source of challenges is, as one would expect, constrained rewriting.

4.2 Rule Removal

HORPO is defined 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 now 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 $\ell \to r\ [\varphi]$. 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 $\Rightarrow$ is rule-monotonic, includes $\to_\kappa$ and stably orients every rewrite rule in $\mathcal{R}_1$, $\to_{\mathcal{R}_1} \subseteq \Rightarrow$. 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 system can be concluded with Lemma 1.

4.3 Constrained HORPO for LCSTRSs

Before adapting HORPO for LCSTRSs, we discuss how the theories may be handled. Let us consider the following system:

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

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 an unconstrained setting, we actually have $n-1 \succ n$. Hence, we must somehow take the logical constraint $n > 0$ into account. To this end, we largely follow the ideas of constrained RPO for first-order LCTRSs [27].

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 $[\![\varphi\sigma]\!] = 1$ implies $[\![\varphi'\sigma]\!] = 1$ for each substitution σ which maps variables in $\mathrm{Var}(\varphi)$ to values. 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 $n > 0 \wedge n > m \models n \sqsupset m$, whereas we cannot have both $m \succ n$ and $n \succ m$ without breaking the well-foundedness of $\succ$. 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 $\sqsupset$. Hence, $\succsim_\varphi$ is *not* necessarily the reflexive closure of $\succ_\varphi$; if it was, 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 constrained HORPO is fully presented, we will show that $\succ_\mathfrak{t}$—the irreflexive relation indexed by the boolean $\mathfrak{t}$—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 (also present in [38]): 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 each 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 [21] 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 $\rightarrow$-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* $\sqsupset_A \colon A \rightarrow A \rightarrow \mathsf{bool}$ *for all* $A \in \mathcal{S}_{\vartheta}$ *such that* $\llbracket \sqsupset_A \rrbracket$ *is a well-founded ordering over* $\mathfrak{X}_A$*. The interpretation* $\llbracket \sqsupseteq_A \rrbracket$ *is assumed to be the reflexive closure of* $\llbracket \sqsupset_A \rrbracket$*. We usually write just* $\sqsupset$ *and* $\sqsupseteq$ *because sorts collapse. Consider* $\llbracket \sqsupset \rrbracket$ *the union* $\bigcup_{A \in \mathcal{S}_{\vartheta}} \llbracket \sqsupset_A \rrbracket$*, and* $\llbracket \sqsupseteq \rrbracket$ *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)* $t = f\ t_1 \cdots t_n$*,* $\mathfrak{s}(f) = \mathfrak{l}$*,* $s_1 \dots s_m \succ_{\varphi}^{\mathfrak{l}} t_1 \dots t_n$*,* $\forall i.\, s \rhd_{\varphi} t_i$*.*
   - *(e)* $t = f\ t_1 \cdots t_n$*,* $\mathfrak{s}(f) = \mathfrak{m}_k$*,* $k \leq n$*,* $s_1 \dots s_{\min(m,k)} \succ_{\varphi}^{\mathfrak{m}} t_1 \dots t_k$*,* $\forall i.\, s \rhd_{\varphi} t_i$*.*
   - *(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* $s \rightarrow_{\kappa}^{*} r$ *and* $t \rightarrow_{\kappa}^{*} r$*.*

Comparison to the Original 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 instead of the traditional monotonicity requirement: if $\succ_\mathfrak{t}$ 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.

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.

This limitation is overcome by means of $\rhd_\varphi$, which actually makes the overall definition more powerful, and is reminiscent of the distinction between $\succ$ and $\succ_{\mathcal{T}_S}$ in later versions of HORPO (e.g., [5]). Other extensions from these works, however, are not yet included in the above definition, 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 orderings: in the original HORPO, $\succsim$ is the reflexive closure of $\succ$, and therefore it suffices to use the more traditional definitions of lexicographic and multiset orderings. Here, as observed above, this would be unnecessarily restrictive.

The split of a single multiset status label in $\mathfrak{m}_2, \mathfrak{m}_3, \ldots$ is due to curried notation—in particular, the possibility of partial application. If we had only a single multiset status label, which would admit, for example, both $f\ 2\ 2 \rhd_\mathfrak{t} f\ 1$ and $f\ 1\ 3 \succ_\mathfrak{t} f\ 2\ 2$, it would be possible that $\succ_\mathfrak{t}$ is not well-founded: note that $g\ (f\ 1) \succ_\mathfrak{t} f\ 1\ 3$ due to, among others, conditions 2b and 3b, and if $f \blacktriangleright g$, we would then have $f\ 2\ 2 \succ_\mathfrak{t} g\ (f\ 1)$ due to, among others, conditions 2b and 3c. This change adds some power to constrained HORPO: we can prove, for example, the termination of the single-rule 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 is not possible if all arguments must be considered, as the original HORPO requires. We do not need $\mathfrak{m}_1$ because this case is already covered by choosing $\mathfrak{l}$.

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 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 $\varphi \models \varphi'$, $\mathfrak{t} \models \varphi'\sigma$ holds 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 $\succ_{\mathsf{t}}$ is compatible with its non-strict counterpart $\succsim_{\mathsf{t}}$; we prove that $\succsim_{\mathsf{t}} ; \succ_{\mathsf{t}} \subseteq \succ_{\mathsf{t}} \cup (\succ_{\mathsf{t}} ; \succ_{\mathsf{t}})$, given the following observation:

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

Proof. By definition, $\succsim_{\mathsf{t}} \supseteq \succ_{\mathsf{t}} \cup \downarrow_\kappa$. We prove $\succsim_{\mathsf{t}} \subseteq \succ_{\mathsf{t}} \cup \downarrow_\kappa$ by induction on the derivation of $s \succsim_{\mathsf{t}} t$. 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 $s \succ_{\mathsf{t}} t$ while the latter implies $s \downarrow_\kappa t$. On the other hand, if s is not a theory term, $s = s_0\ s_1$, $t = t_0\ t_1$, $s_0 \succsim_{\mathsf{t}} t_0$ and $s_1 \succsim_{\mathsf{t}} t_1$, by induction, if $s_0 \succ_{\mathsf{t}} t_0$ or $s_1 \succ_{\mathsf{t}} t_1$, we can prove $s \succ_{\mathsf{t}} t$ in the same manner as we prove the rule-monotonicity of $\succ_{\mathsf{t}}$, or $s_0 \downarrow_\kappa t_0$ and $s_1 \downarrow_\kappa t_1$, then $s \downarrow_\kappa t$. □

Theorem 4 plays an important role in the well-foundedness proof of $\succ_{\mathsf{t}}$ as well.

For the compatibility of $\succ_{\mathsf{t}}$ with $\succsim_{\mathsf{t}}$, it remains to prove that $\downarrow_\kappa ; \succ_{\mathsf{t}} \subseteq \succ_{\mathsf{t}}$, which is implied by the following lemma:

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

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

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

The compatibility follows as a corollary:

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

Well-Foundedness. Following [21], we base the well-foundedness proof of $\succ_{\mathsf{t}}$ on the predicate of computability [40,17]. There are, however, two major differences, which pose new technical challenges: $\succsim_{\mathsf{t}}$ is no more the reflexive closure of $\succ_{\mathsf{t}}$ and curried notation instead of uncurried notation is in use.

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

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

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

We refer to [41] for the proof of the following property of $\succ_{\mathsf{t}}^{\mathsf{m}}$:

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

Proof. See Theorem 3.7 in [41]. □

In comparison to [41], 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 $\succsim_{\mathsf{t}}$ is not necessarily transitive due to its inclusion of $\succ_{\mathsf{t}}$. Fortunately, one observation resolves this issue: $\succ_{\mathsf{t}}^{\mathsf{m}}$ can equivalently be seen as induced by $\downarrow_\kappa$ and $\succ_{\mathsf{t}}$ due to Theorem 4. In the same spirit, we can prove the following property:

Theorem 7. $\downarrow_\kappa^{\mathsf{m}} ; \succ_{\mathsf{t}}^{\mathsf{m}} \subseteq \succ_{\mathsf{t}}^{\mathsf{m}}$ *where $s_1 \dots s_n \downarrow_\kappa^{\mathsf{m}} t_1 \dots t_n$ 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 [41]. □

Our definition of computability (or reducibility [17]) 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 $\succ_{\mathsf{t}}$, 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 [21], a term is called neutral if it is not a λ-abstraction. Due to the exclusion of λ-abstractions, 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 one-step reduct (with respect to $\succ_{\mathsf{t}}$) 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 of curried notation, neutral LCSTRS 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 $s \succ_{\mathsf{t}} t$, if s is computable, so is t.*
2. *All computable terms are terminating with respect to $\succ_{\mathsf{t}}$.*
3. *Given a neutral term s, if t is computable for all t such that $s \succ_{\mathsf{t}} t$, 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 [21], 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 $\bar{t} = t_1 \ldots t_n$, 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* $(\bar{s})_1 \ldots (\bar{s})_{\mathrm{ar}(f)} \succ_{\mathfrak{t}}^{\mathfrak{l}} (\bar{t})_1 \ldots (\bar{t})_{\mathrm{ar}(f)}$, *or*
- $\mathfrak{s}(f) = \mathfrak{m}_k$ *and*
  - $(\bar{s})_1 \ldots (\bar{s})_k \succ_{\mathfrak{t}}^{\mathfrak{m}} (\bar{t})_1 \ldots (\bar{t})_k$, *or*
  - $(\bar{s})_1 \ldots (\bar{s})_k \downarrow_\kappa^{\mathfrak{m}} (\bar{t})_1 \ldots (\bar{t})_k$, $\forall i > k.\, (\bar{s})_i \succsim_{\mathfrak{t}} (\bar{t})_i$ *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 $\llbracket \sqsupset \rrbracket$. 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 $t = f\ t_1 \cdots t_n$, $\mathfrak{s}(f) = \mathfrak{l}$, $s_1 \ldots s_m \succ_{\mathfrak{t}}^{\mathfrak{l}} t_1 \ldots t_n$ 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 $t = f\ t_1 \cdots t_n$, $\mathfrak{s}(f) = \mathfrak{m}_k$, $k \leq n$, $s_1 \ldots s_{\min(m,k)} \succ_{\mathfrak{t}}^{\mathfrak{m}} t_1 \ldots t_k$ 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 $(s_1 \ldots s_m)_i = (s_1 \ldots s_{m+1})_i$ for all $i \leq m$ and $(s_1 \ldots s_m)_{m+1} = \top \succ_{\mathfrak{t}} (s_1 \ldots s_{m+1})_{m+1}$. 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 Discussion: HORPO and Dependency Pairs

In Section 4, we discussed rule removal, and presented a reduction ordering to prove termination. However, in practice it is not so common to directly use reduction orderings as a termination method. Rather, the norm in the literature nowadays is to use dependency pairs.

The dependency pair framework [1,16] allows a single term rewriting system to be split into multiple "DP problems", each of which can then be analyzed independently. The framework operates by iteratively simplifying DP problems until none remain, in which case the original system is proved terminating. There are variants for many styles of term rewriting, including first-order LCTRSs [25] and unconstrained higher-order TRSs [39,25,11].

Importantly, many existing techniques can be reformulated as “processors” (DP problem simplifiers) in the dependency pair framework. Such techniques include reduction orderings, which are at the heart of the dependency pair framework. This combination is far more powerful than using reduction orderings directly because the monotonicity requirement is replaced by weak monotonicity, and we do not have to orient the entire system in one go.

Consider the following first-order LCTRS:

$$\begin{aligned} &\mathsf{u}\ x\ y \to \mathsf{u}\ (x+1)\ (y*2) \quad [x < 100] \qquad && \mathsf{v}\ y \to \mathsf{v}\ (y-1) \quad [y > 0] \\ &\mathsf{u}\ 100\ y \to \mathsf{v}\ y && \end{aligned}$$

This system cannot be handled by HORPO directly: the ordering $[\![\sqsupset_{\mathsf{int}}]\!]$ needs to be fixed globally, so we can either orient the rewrite rule at the top-left corner or the one at the top-right corner, but not both at the same time. We could address this dilemma by using a more elaborate definition of HORPO (for example, by giving every function symbol an additional status that indicates the theory ordering to be used for each of its arguments), but this seems redundant: in practice, such a system would be handled by the dependency pair framework. Following the definition in [25], the above system would be split in two separate DP problems corresponding to the two loops:

$$\left\{\, \mathsf{u}^\sharp\ x\ y \to \mathsf{u}^\sharp\ (x+1)\ (y*2) \quad [x < 100] \,\right\} \qquad \left\{\, \mathsf{v}^\sharp\ y \to \mathsf{v}^\sharp\ (y-1) \quad [y > 0] \,\right\}$$

which could then be handled independently.

While dependency pairs for LCSTRSs are not yet defined (and beyond the scope of this paper), we postulate that the definitions for curried higher-order rewriting in [11] and first-order constrained rewriting in [25] can be combined in a natural way. In this setting, HORPO would naturally be combined with *argument filterings* [1,11]. That is, since we only require *weak* monotonicity, some arguments can be removed. For example, the first DP problem above can be handled by showing the following inequalities:

$$\mathsf{u}^\sharp\ x \succ_{x<100} \mathsf{u}^\sharp\ (x+1) \qquad \mathsf{u} \succsim_{x<100} \mathsf{u} \qquad \mathsf{v} \succsim_{y>0} \mathsf{v} \qquad \mathsf{u} \succsim_{\mathfrak{t}} \mathsf{v}$$

This is the case with $\mathsf{u} \blacktriangleright \mathsf{v}$.

6 Implementation

A preliminary implementation of LCSTRSs is available in Cora through the link:

https://github.com/hezzel/cora

Cora is an open-source analyzer for constrained rewriting, which can be used both as a stand-alone tool and as a library. Note that Cora is still in active development, and its functionalities, as well as its interface, are subject to change.

Nevertheless, Cora is already used in several student projects. Cora supports only the theories of integers and booleans so far, but is intended to eventually support any theory, provided that an SMT solver is able to handle it. Example input files are supplied in the above repository. The version of this paper is available in [28].

Automating Constrained HORPO. Cora includes an implementation of constrained HORPO. Following existing termination tools such as AProVE [14], NaTT [42] and Wanda [26], we use an SMT encoding such that a satisfying assignment to variables in the SMT problem corresponds to a combination of the precedence $\blacktriangleright$, the status $\mathfrak{s}$ and the ordering $[\![\sqsupset_{\mathsf{int}}]\!]$ that proves the termination of the encoded system by constrained HORPO. As for booleans, we simply choose the ordering $[\![\sqsupset_{\mathsf{bool}}]\!]$ such that $[\![\mathfrak{t} \sqsupset_{\mathsf{bool}} \mathfrak{f}]\!] = 1$.

To encode the precedence and the status, we introduce integer variables $\mathtt{prec}_f$ and $\mathtt{stat}_f$ for each function symbol f that is not a value. We require that $\mathtt{prec}_f < 0$ if f is a theory symbol, and that $\mathtt{prec}_f \geq 0$ otherwise—so that $\mathtt{prec}_f > \mathtt{prec}_g$ corresponds to $f \blacktriangleright g$. The value k of $\mathtt{stat}_f$ indicates $\mathfrak{s}(f) = \mathfrak{l}$ if $k = 1$, and $\mathfrak{s}(f) = \mathfrak{m}_k$ if $k > 1$. We let $\mathtt{down}$ be a boolean variable which indicates the choice between two possibilities for $[\![\sqsupset_{\mathsf{int}}]\!]$: $\lambda m.\, \lambda n.\, m > -M \wedge m > n$ and $\lambda m.\, \lambda n.\, m < M \wedge m < n$ (the choice of M is discussed below).

In the derivation of $s \succ_{\varphi} t$, all assertions assume the form $s'\ R_{\varphi}\ t'$ where s' and t' are subterms of s and t, respectively (see Example 9). Hence, given a finite set of rewrite rules, there are only finitely many possible assertions to be analyzed. By inspecting the definition of constrained HORPO, we also note that there are no cyclic dependencies. For all $\ell \to r\ [\varphi]$, respective subterms s and t of ℓ and r, and $R \in \{\succsim, \succ, \rhd, 1\mathrm{a}, 1\mathrm{b}, \ldots, 3\mathrm{f}\}$, we thus introduce a variable $\langle s\ R_{\varphi}\ t\rangle$ with its defining constraint. Without going into detail for all the cases, we provide a few key examples:

- If s and t do not have equal types, we add $\neg\langle s \succsim_{\varphi} t\rangle$; otherwise, we add $\langle s \succsim_{\varphi} t\rangle \implies \langle s\ 1\mathrm{a}_{\varphi}\ t\rangle \vee \langle s\ 1\mathrm{b}_{\varphi}\ t\rangle \vee \langle s\ 1\mathrm{c}_{\varphi}\ t\rangle \vee \langle s\ 1\mathrm{d}_{\varphi}\ t\rangle$, which states that if $s \succsim_{\varphi} t$ holds, it must hold in one of the defining cases 1a, 1b, 1c and 1d. Each of these cases in turn has its defining constraint.
- $\langle f\ s_1 \cdots s_m\ 3\mathrm{c}_{\varphi}\ g\ t_1 \cdots t_n\rangle \implies \mathtt{prec}_f > \mathtt{prec}_g \wedge \bigwedge_j \langle f\ s_1 \cdots s_m \rhd_{\varphi} t_j\rangle$.
- We come up with the defining constraint of $\langle s\ 2\mathrm{a}_{\varphi}\ t\rangle$ by case analysis:
  - If either of s and t is not a theory term, or their respective types are not the same theory sort, or $\mathrm{Var}(s) \cup \mathrm{Var}(t) \nsubseteq \mathrm{Var}(\varphi)$, we add $\neg\langle s\ 2\mathrm{a}_{\varphi}\ t\rangle$.
  - Otherwise, we consider the type of s and t:
    - The type is int. We respectively check if $\varphi \implies s > -M \wedge s > t$ and $\varphi \implies s < M \wedge s < t$ are valid. If the former is *not* valid, we add $\langle s\ 2\mathrm{a}_{\varphi}\ t\rangle \implies \neg\mathtt{down}$; if the latter is *not* valid, we add $\langle s\ 2\mathrm{a}_{\varphi}\ t\rangle \implies \mathtt{down}$. That is, if both of the validity checks fail, both of the constraints are added, which is equivalent to adding $\neg\langle s\ 2\mathrm{a}_{\varphi}\ t\rangle$.
    - The type is bool. We add $\neg\langle s\ 2\mathrm{a}_{\varphi}\ t\rangle$ if $\varphi \implies s \wedge \neg t$ is not valid; if it is valid, nothing is added and the SMT solver is free to set true for the variable $\langle s\ 2\mathrm{a}_{\varphi}\ t\rangle$.

Here M is twice the largest absolute value of integers occurring in the rewrite rules, or just 1000 if that is too large—this value is chosen arbitrarily. Note that the validity checks are *not* included as part of the SMT problem: if they were included, the satisfiability problem would contain universal quantification, which is typically hard to solve. We rather pose a separate question to the SMT solver every time we encounter theory comparison, and for integers, consider whether the pair can be oriented downward with $\lambda m.\, \lambda n.\, m > -M \wedge m > n$, upward with $\lambda m.\, \lambda n.\, m < M \wedge m < n$, or not at all. Hence, we must fix the bound M beforehand.

- The hardest is 3e: we need not only to encode the multiset comparison, but also to make sure that only k arguments are to be considered on both sides (should there be more). Following Definition 4, we introduce boolean variables $\texttt{strict}_1, \ldots, \texttt{strict}_m$ where $\texttt{strict}_i$ indicates $i \in I$, and integer variables $\pi(1), \ldots, \pi(n)$. The defining constraint of $\langle f\ s_1 \cdots s_m\ 3\mathrm{e}_\varphi\ f\ t_1 \cdots t_n \rangle$ is the conjunction of the following components:
  - $\langle f\ s_1 \cdots s_m\ 3\mathrm{e}_\varphi\ f\ t_1 \cdots t_n \rangle \implies 2 \le \texttt{stat}_f \le n$.
  - $\langle f\ s_1 \cdots s_m\ 3\mathrm{e}_\varphi\ f\ t_1 \cdots t_n \rangle \implies \bigwedge_j \langle f\ s_1 \cdots s_m \rhd_\varphi t_j \rangle$.
  - $\langle f\ s_1 \cdots s_m\ 3\mathrm{e}_\varphi\ f\ t_1 \cdots t_n \rangle \implies \bigvee_i \texttt{strict}_i$.
  - For all $i \in \{\, 1, \ldots, m \,\}$, $\langle f\ s_1 \cdots s_m\ 3\mathrm{e}_\varphi\ f\ t_1 \cdots t_n \rangle \wedge \texttt{strict}_i \implies i \le \texttt{stat}_f$. That is, $I \subseteq \{\, 1, \ldots, k \,\}$ if $\mathfrak{s}(f) = \mathfrak{m}_k$.
  - For all $j \in \{\, 1, \ldots, n \,\}$, $\langle f\ s_1 \cdots s_m\ 3\mathrm{e}_\varphi\ f\ t_1 \cdots t_n \rangle \wedge j \le \texttt{stat}_f \implies 1 \le \pi(j) \wedge \pi(j) \le m \wedge \pi(j) \le \texttt{stat}_f$. That is, $1 \le \pi(j) \le \min(m, k)$ for all $j \in \{\, 1, \ldots, k \,\}$ if $\mathfrak{s}(f) = \mathfrak{m}_k$.
  - For all $i \in \{\, 1, \ldots, m \,\}$, $j \in \{\, 1, \ldots, n-1 \,\}$ and $j' \in \{\, j+1, \ldots, n \,\}$, $\langle f\ s_1 \cdots s_m\ 3\mathrm{e}_\varphi\ f\ t_1 \cdots t_n \rangle \implies \texttt{strict}_i \vee \pi(j) \ne i \vee \pi(j') \ne i$. That is, $\left|\pi^{-1}(i)\right| \le 1$ for all $i \in \{\, 1, \ldots, m \,\} \setminus I$—which suffices because we can add to I all $i \in \{\, 1, \ldots, \min(m, k) \,\} \setminus I$ such that $\left|\pi^{-1}(i)\right| = 0$ without changing the generalized multiset ordering if $\mathfrak{s}(f) = \mathfrak{m}_k$.
  - For all $i \in \{\, 1, \ldots, m \,\}$ and $j \in \{\, 1, \ldots, n \,\}$, $\langle f\ s_1 \cdots s_m\ 3\mathrm{e}_\varphi\ f\ t_1 \cdots t_n \rangle \wedge \pi(j) = i \wedge \texttt{strict}_i \implies \langle s_i \succ_\varphi t_j \rangle$.
  - For all $i \in \{\, 1, \ldots, m \,\}$ and $j \in \{\, 1, \ldots, n \,\}$, $\langle f\ s_1 \cdots s_m\ 3\mathrm{e}_\varphi\ f\ t_1 \cdots t_n \rangle \wedge \pi(j) = i \wedge \neg\texttt{strict}_i \implies \langle s_i \succsim_\varphi t_j \rangle$.

Cora succeeds in proving that all the examples in this paper are terminating, except Example 2, which is non-terminating.

7 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 is LCTRSs [27,12], the first-order formalism for constrained rewriting upon which the present work is built. Similarly, there are numerous formalisms for higher-order term rewriting, but without built-in logical constraints, e.g., [21,22,31]. It seems likely that the methods for analyzing those can be extended with support for SMT, as what is done for HORPO in this paper.

Also worth mentioning is the K Framework [35], which, like our formalism, can be used as an intermediate language for program analysis and is based on a form of first-order rewriting. The K tool includes techniques through *reachability logic*, rather than methods like HORPO.

There are several works that analyze functional programs using term rewriting, e.g., [2,15]. However, they typically use translations to first-order systems. Hence, some of the structure of the initial problem is lost, and their power is weakened.

HORPO. Our definition of constrained HORPO is based on the first-order constrained RPO for LCTRSs [27] and the first definition of higher-order RPO [21]. There have been other HORPO extensions since, e.g., [5,6], and we believe that the ideas for these extensions can also be applied to constrained HORPO. We have not done so because the purpose of this paper is to show *that* and *how* techniques for analyzing higher-order systems extend, not to introduce the most powerful (and consequently more elaborate) ones.

Also worth mentioning is [4], a higher-order RPO for λ-free systems. This variant is defined for the purpose of superposition rather than termination analysis, and is ground-total but generally not monotonic.

Functional Programming. There are many works performing direct analyses of functional programs, including termination analysis, although they typically concern specific programming languages such as Haskel (e.g., [19]) and OCaml (e.g., [20]). A variety of techniques have been proposed, such as sized types [33] and decreasing measures on data [18], but as far as we can find, there is no real parallel of many rewriting techniques such as RPO. We hope that, through LCSTRSs, we can help make the techniques of term rewriting available to the functional programming community.

8 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 logical constraints. Despite being a very basic method, this is already powerful enough to handle examples in this paper. Both LCSTRSs and constrained HORPO are implemented in our new analysis tool Cora.

In the future, we intend to extend more techniques, both first-order and higher-order, to this formalism, and to implement them in 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 program analysis.

A natural starting point is to increase our power in termination analysis by extending dependency pairs [1,39,11,25] and various supporting methods like the subterm criterion and usable rules. In addition, methods for analyzing complexity, reachability and equivalence (e.g., through rewriting induction [34,12]), which have been defined for first-order LCTRSs, are natural directions for higher-order extension as well.

Acknowledgments. The authors are supported by NWO VI.Vidi.193.075, project "CHORPE". We thank Deivid Vale for his work on Cora and his assistance in the preparation of the artifact, Carsten Fuhs for his comments on an early draft of this paper, and the anonymous reviewers for their helpful feedback.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this paper.

References

1. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. TCS **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: Reppy, J. (ed.) Proc. ICFP. pp. 152–164 (2015). https://doi.org/10.1145/2784731.2784753
3. Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB). https://smtlib.cs.uiowa.edu
4. Blanchette, J.C., Waldmann, U., Wand, D.: A lambda-free higher-order recursive path order. In: Esparza, J., Murawski, A.S. (eds.) Proc. FoSSaCS. pp. 461–479 (2017). https://doi.org/10.1007/978-3-662-54458-7_27
5. Blanqui, F., Jouannaud, J.P., Rubio, A.: HORPO with computability closure: a reconstruction. In: Dershowitz, N., Voronkov, A. (eds.) Proc. LPAR. pp. 138–150 (2007). https://doi.org/10.1007/978-3-540-75560-9_12
6. Blanqui, F., Jouannaud, J.P., Rubio, A.: The computability path ordering: the end of a quest. In: Kaminski, M., Martini, S. (eds.) Proc. CSL. pp. 1–14 (2008). https://doi.org/10.1007/978-3-540-87531-4_1
7. Ciobâcă, Ş., Lucanu, D., Buruiană, A.S.: Operationally-based program equivalence proofs using LCTRSs. JLAMP **135**, 100894:1–100894:22 (2023). https://doi.org/10.1016/j.jlamp.2023.100894
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. 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. 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 TOPLAS **33**(2), 7:1–7:39 (2011). https://doi.org/10.1145/1890028.1890030
16. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: combining techniques for automated termination proofs. In: Baader, F., Voronkov, A. (eds.) Proc. LPAR. pp. 301–331 (2005). https://doi.org/10.1007/978-3-540-32275-7_21
17. Girard, J.Y., Taylor, P., Lafont, Y.: Proofs and Types. Cambridge University Press (1989)
18. Hamza, J., Voirol, N., Kunčak, V.: System FR: formalized foundations for the Stainless verifier. PACMPL **3**(OOPSLA), 166:1–166:30 (2019). https://doi.org/10.1145/3360592
19. Handley, M.A.T., Vazou, N., Hutton, G.: Liquidate your assets: reasoning about resource usage in Liquid Haskell. PACMPL **4**(POPL), 24:1–24:27 (2019). https://doi.org/10.1145/3371092
20. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ML. In: Madhusudan, P., Seshia, S.A. (eds.) Proc. CAV. pp. 781–786 (2012). https://doi.org/10.1007/978-3-642-31424-7_64
21. 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
22. Klop, J.W., van Oostrom, V., van Raamsdonk, F.: Combinatory reduction systems: introduction and survey. TCS **121**(1–2), 279–308 (1993). https://doi.org/10.1016/0304-3975(93)90091-7
23. Kojima, M., Nishida, N.: From starvation freedom to all-path reachability problems in constrained rewriting. In: Hanus, M., Inclezan, D. (eds.) Proc. PADL. pp. 161–179 (2023). https://doi.org/10.1007/978-3-031-24841-2_11
24. Kojima, M., Nishida, N.: Reducing non-occurrence of specified runtime errors to all-path reachability problems of constrained rewriting. JLAMP **135**, 100903:1–100903:19 (2023). https://doi.org/10.1016/j.jlamp.2023.100903
25. Kop, C.: Termination of LCTRSs. In: Waldmann, J. (ed.) Proc. WST. pp. 59–63 (2013). https://doi.org/10.48550/arXiv.1601.03206
26. Kop, C.: WANDA — a higher order termination tool. In: Ariola, Z.M. (ed.) Proc. FSCD. pp. 36:1–36:19 (2020). https://doi.org/10.4230/LIPIcs.FSCD.2020.36
27. 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
28. Kop, C., Vale, D.: hezzel/cora artifact: ESOP2024 release v4 (2024). https://doi.org/10.5281/zenodo.10560907
29. Kusakari, K.: On proving termination of term rewriting systems with higher-order variables. IPSJ Trans. Program. **42**(SIG 7), 35–45 (2001), http://id.nii.ac.jp/1001/00016864/

30. Nagao, T., Nishida, N.: Rewriting induction for constrained inequalities. SCP **155**, 76–102 (2018). https://doi.org/10.1016/j.scico.2017.10.012
31. Nipkow, T.: Higher-order critical pairs. In: Kahn, G. (ed.) Proc. LICS. pp. 342–349 (1991). https://doi.org/10.48456/tr-218
32. Nishida, N., Winkler, S.: Loop detection by logically constrained term rewriting. In: Piskac, R., Rümmer, P. (eds.) Proc. VSTTE. pp. 309–321 (2018). https://doi.org/10.1007/978-3-030-03592-1_18
33. Pareto, L.: Sized types (1998), licentiate thesis. Chalmers University of Technology
34. Reddy, U.S.: Term rewriting induction. In: Stickel, M.E. (ed.) Proc. CADE. pp. 162–177 (1990). https://doi.org/10.1007/3-540-52885-7_86
35. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. JLAP **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
36. 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
37. 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
38. Schöpf, J., Middeldorp, A.: Confluence criteria for logically constrained rewrite systems. In: Pientka, B., Tinelli, C. (eds.) Proc. CADE. pp. 474–490 (2023). https://doi.org/10.1007/978-3-031-38499-8_27
39. Suzuki, S., Kusakari, K., Blanqui, F.: Argument filterings and usable rules in higher-order rewrite systems. IPSJ Online Trans. **4**, 114–125 (2011). https://doi.org/10.2197/ipsjtrans.4.114
40. Tait, W.W.: Intensional interpretations of functionals of finite type I. JSL **32**(2), 198–212 (1967). https://doi.org/10.2307/2271658
41. 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
42. Yamada, A., Kusakari, K., Sakabe, T.: Nagoya termination tool. In: Dowek, G. (ed.) Proc. RTA–TLCA. pp. 466–475 (2014). https://doi.org/10.1007/978-3-319-08918-8_32