Higher-Order Constrained Dependency Pairs for (Universal) Computability

Liye Guo
Radboud University, Nijmegen, The Netherlands

Kasper Hagens
Radboud University, Nijmegen, The Netherlands

Cynthia Kop
Radboud University, Nijmegen, The Netherlands

Deivid Vale
Radboud University, Nijmegen, The Netherlands

Abstract

Dependency pairs constitute a series of very effective techniques for the termination analysis of term rewriting systems. In this paper, we adapt the static dependency pair framework to logically constrained simply-typed term rewriting systems (LCSTRSs), a higher-order formalism with logical constraints built in. We also propose the concept of universal computability, which enables a form of open-world termination analysis through the use of static dependency pairs.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting

Keywords and phrases Higher-order term rewriting, constrained rewriting, dependency pairs

Digital Object Identifier 10.4230/LIPIcs.MFCS.2024.57

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2406.19379` [11]

Supplementary Material *Software (Source Code)*: `https://github.com/hezzel/cora` [20]
Software (Source Code): `https://zenodo.org/records/12551027` [21]

Funding The authors are supported by NWO VI.Vidi.193.075, project "CHORPE".

1 Introduction

Logically constrained simply-typed term rewriting systems (LCSTRSs) [12] are a formalism of higher-order term rewriting with logical constraints (built on its first-order counterpart [19]). Proposed for program analysis, LCSTRSs offer a flexible representation of programs owing to – in contrast with traditional TRSs – their support for primitive data types such as (arbitrary-precision or fixed-width) integers and floating-point numbers. Without compromising the ability to directly reason about these widely used data types, LCSTRSs bridge the gap between the abundant techniques based on term rewriting and automatic program analysis.

We consider *termination* analysis in this paper. The termination of LCSTRSs was first discussed in [12] through a variant of the higher-order recursive path ordering (HORPO) [14]. This paper furthers that discussion by introducing dependency pairs [1] to LCSTRSs. As a broad framework for termination, this method was initially proposed for unconstrained first-order term rewriting, and was later generalized in a variety of higher-order settings (see, e.g., [31, 23, 30, 2]). Modern termination analyzers rely heavily on dependency pairs.

In higher-order termination analysis, dependency pairs take two forms: the dynamic [31, 23] and the static [30, 2, 24, 7]. This paper concentrates on *static* dependency pairs, and is based on the definitions in [7, 24]. First-order dependency pairs with logical constraints have been informally defined by the third author [15], from which we also take inspiration.

For program analysis, the traditional notion of termination can be inefficient, and arguably insufficient. It assumes that the whole program is known and analyzed, i.e., *closed-world* analysis. This way even small programs that happen to import a large standard library need sophisticated analysis; local changes in a multipart, previously verified program also require the entire analysis to be redone. As O'Hearn [26] argues (in a different context), studying *open-world* analysis opens up many applications. In particular, it is practically desirable to analyze the termination of standard libraries – or modules of a larger program in general – without prior knowledge of how the functions they define may be used.

It is tricky to characterize such a property, especially in the presence of higher-order arguments. For example, map and fold are usually considered "terminating", even though passing a non-terminating function to them can surely result in non-termination. Hence, we need to narrow our focus to certain "reasonable" calls. On the other hand, the function $\mathsf{app}\ (\mathsf{lam}\ f) \to f$ where $\mathsf{app} : \mathsf{o} \to \mathsf{o} \to \mathsf{o}$ and $\mathsf{lam} : (\mathsf{o} \to \mathsf{o}) \to \mathsf{o}$ would generally be considered "non-terminating", because if we define $\mathsf{w}\ x \to \mathsf{app}\ x\ x$, an infinite rewrite sequence starts from $\mathsf{app}\ (\mathsf{lam}\ \mathsf{w})\ (\mathsf{lam}\ \mathsf{w})$ – this encodes the famous Ω in the untyped λ-calculus. The property we are looking for must distinguish map and fold from app.

To capture this property, we propose a new concept, called *universal computability.* In light of information hiding, this concept can be further generalized to *public computability.* We will see that static dependency pairs are a natural vehicle for analyzing these properties.

Various modular aspects of term rewriting have been studied by the community. Our scenario roughly corresponds to hierarchical combinations [27, 28, 29, 6], where different parts of a program are analyzed separately. We follow this terminology so that it will be easier to compare our work with the literature. However, our setup – higher-order constrained rewriting – is separate from the first-order and unconstrained setting in which hierarchical combinations were initially proposed. Furthermore, our approach has a different focus – namely, the use of static dependency pairs.

Contributions. We recall the formalism of LCSTRSs and the predicate of computability in Section 2. Then the contributions of this paper follow:

- We propose in Section 3 the first definition of *dependency pairs* for higher-order logically constrained TRSs. This is also the first DP approach for constrained rewriting as the prior work on first-order constrained dependency pairs [15] has never been formally published.
- We define in Section 4 the *constrained DP framework* for termination analysis with five classes of *DP processors*, which can be used to simplify termination problems.
- We extend the notion of a *hierarchical combination* [27, 28, 29, 6] to LCSTRSs and define *universal* and *public computability* in Section 5. We also fine-tune the DP framework to support these properties, and provide extra DP processors for public computability. This allows the DP framework to be used for open-world analysis. We base Section 5 on LCSTRSs to emphasize those notions in real-world programming, but they are new and of theoretical interest in higher-order term rewriting even without logical constraints.
- We have implemented the DP framework for both termination and public computability in our open-source analyzer Cora. We describe the experimental evaluation in Section 6.

2 Preliminaries

In this section, we collect the preliminary definitions and results we need from the literature. First, we recall the definition of an LCSTRS [12]. In this paper, we put a restriction on rewrite rules: ℓ is always a pattern in $\ell \to r\ [\varphi]$. Next, we recall the definition of computability (with accessibility) from [7]. This version is particularly tailored for static dependency pairs.

2.1 Logically Constrained STRSs

Terms Modulo Theories. 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. 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 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 $T(\mathcal{F}, \mathcal{V})$ of *terms* over $\mathcal{F}$ and $\mathcal{V}$ consists of pre-terms that have a type. We write $t : A$ if a term t has type A. We assume there are infinitely many variables of each type.

The set $\mathrm{Var}(t)$ of variables in $t \in T(\mathcal{F}, \mathcal{V})$ is defined by $\mathrm{Var}(f) = \emptyset$ 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 $\mathrm{Var}(t) = \emptyset$.

For constrained rewriting, we make further assumptions. 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 a theory type is essentially a non-empty list of theory sorts. 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. Theory symbols whose type is a theory sort are called *values*. Elements of $T(\mathcal{F}_\vartheta, \mathcal{V})$ are called *theory terms*.

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 if $A \in \mathcal{S}_\vartheta$. 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 write just $\llbracket\cdot\rrbracket$ when the type can be deduced.

▶ **Example 1.** 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 $\{ - \} \cup \mathbb{Z}$ where $- : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ and $n : \mathsf{int}$ for all $n \in \mathbb{Z}$. The values are the elements of $\mathbb{Z}$. Let $\mathfrak{X}_{\mathsf{int}}$ be $\mathbb{Z}$, $\llbracket\cdot\rrbracket_{\mathsf{int}}$ be the identity mapping and $\llbracket - \rrbracket$ be the mapping $\lambda m.\, \lambda n.\, m - n$. The interpretation of $(-)\ 1$ is the mapping $\lambda n.\, 1 - n$.

Substitutions, Contexts and Subterms. 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 $(t_0\ t_1)\sigma = (t_0\sigma)\ (t_1\sigma)$. Let $[x_1 := t_1, \ldots, x_n := t_n]$ denote the substitution σ such that $\sigma(x_i) = t_i$ for all i, and $\sigma(y) = y$ for all $y \in \mathcal{V} \setminus \{ x_1, \ldots, x_n \}$.

A context is a term containing a hole. Let $\Box$ be a special terminal symbol and assign to it a type A; a *context* $C[]$ is an element of $T(\mathcal{F}, \mathcal{V} \cup \{ \Box \})$ such that $\Box$ occurs in $C[]$ exactly once. Given a term $t : A$, let $C[t]$ denote the term produced by replacing $\Box$ in $C[]$ with t.

A term t is called a (maximally applied) *subterm* of a term s, written as $s \trianglerighteq t$, if either $s = t$, $s = s_0\ s_1$ where $s_1 \trianglerighteq t$, or $s = s_0\ s_1$ where $s_0 \trianglerighteq t$ and $s_0 \neq t$; i.e., $s = C[t]$ for $C[]$ that is not of form $C'[\Box\ t_1]$. We write $s \triangleright t$ and call t a *proper subterm* of s if $s \trianglerighteq t$ and $s \neq t$.

Constrained Rewriting. Constrained rewriting requires the theory sort bool: we henceforth assume that $\mathsf{bool} \in \mathcal{S}_\vartheta$, $\{ \mathfrak{f}, \mathfrak{t} \} \subseteq \mathcal{F}_\vartheta$, $\mathfrak{X}_{\mathsf{bool}} = \{ 0, 1 \}$, $\llbracket \mathfrak{f} \rrbracket_{\mathsf{bool}} = 0$ and $\llbracket \mathfrak{t} \rrbracket_{\mathsf{bool}} = 1$. A *logical constraint* is a theory term φ such that φ has type bool and the type of each variable in $\mathrm{Var}(\varphi)$

is a theory sort. A (constrained) *rewrite rule* is a triple $\ell \to r\ [\varphi]$ where ℓ and r are terms which have the same type, φ is a logical constraint, the type of each variable in $\mathrm{Var}(r) \setminus \mathrm{Var}(\ell)$ is a theory sort and ℓ is a pattern that takes the form $f\ t_1 \cdots t_n$ for some function symbol f and contains at least one function symbol in $\mathcal{F} \setminus \mathcal{F}_\vartheta$. Here a *pattern* is a term whose subterms are either $f\ t_1 \cdots t_n$ for some function symbol f or a variable. A substitution σ is said to *respect* $\ell \to r\ [\varphi]$ if $\sigma(x)$ is a value for all $x \in \mathrm{Var}(\varphi) \cup (\mathrm{Var}(r) \setminus \mathrm{Var}(\ell))$ and $[\![\varphi\sigma]\!] = 1$.

A *logically constrained simply-typed term rewriting system* (LCSTRS) collects the above data – $\mathcal{S}$, $\mathcal{S}_\vartheta$, $\mathcal{F}$, $\mathcal{F}_\vartheta$, $\mathcal{V}$, $(\mathfrak{X}_A)$ and $[\![\cdot]\!]$ – along with a set $\mathcal{R}$ of rewrite rules. We usually let $\mathcal{R}$ alone stand for the system. The set $\mathcal{R}$ induces the *rewrite relation* $\to_\mathcal{R}$ over terms: $t \to_\mathcal{R} t'$ if and only if there exists a context $C[]$ such that either (1) $t = C[\ell\sigma]$ and $t' = C[r\sigma]$ for some rewrite rule $\ell \to r\ [\varphi] \in \mathcal{R}$ and some substitution σ which respects $\ell \to r\ [\varphi]$, or (2) $t = C[f\ v_1 \cdots v_n]$ and $t' = C[v']$ for some theory symbol f and some values $v_1, \ldots, v_n, v'$ with $n > 0$ and $[\![f\ v_1 \cdots v_n]\!] = [\![v']\!]$. When no ambiguity arises, we may write $\to$ for $\to_\mathcal{R}$.

If $t \to_\mathcal{R} t'$ due to the second condition above, we also write $t \to_\kappa t'$ and call it a *calculation step*. Theory symbols that are not a value are called *calculation symbols*. Let $t\!\downarrow_\kappa$ denote the (unique) κ-normal form of t, i.e., the term t' such that $t \to_\kappa^* t'$ and $t' \not\to_\kappa t''$ for any t''. For example, $(f\ (7 * (3 * 2)))\!\downarrow_\kappa = f\ 42$ if f is not a calculation symbol, or if $f : \mathsf{int} \to A \to B$.

A rewrite rule $\ell \to r\ [\varphi]$ is said to *define* a function symbol f if $\ell = f\ t_1 \cdots t_n$. Given an LCSTRS $\mathcal{R}$, f is called a *defined symbol* if some rewrite rule in $\mathcal{R}$ defines f. Let $\mathcal{D}$ denote the set of defined symbols. Values and function symbols in $\mathcal{F} \setminus (\mathcal{F}_\vartheta \cup \mathcal{D})$ are called *constructors*.

▶ **Example 2.** Below is the factorial function in continuation-passing style as an LCSTRS:

$$\begin{array}{lll} \mathsf{fact}\ n\ k \to k\ 1 & [n \le 0] & \mathsf{comp}\ g\ f\ x \to g\ (f\ x) \\ \mathsf{fact}\ n\ k \to \mathsf{fact}\ (n-1)\ (\mathsf{comp}\ k\ ((*)\ n)) & [n > 0] & \mathsf{id}\ x \to x \end{array}$$

We use infix notation for some binary operators, and omit the logical constraint of a rewrite rule when it is $\mathfrak{t}$. An example rewrite sequence is $\mathsf{fact}\ 1\ \mathsf{id} \to \mathsf{fact}\ (1-1)\ (\mathsf{comp}\ \mathsf{id}\ ((*)\ 1)) \to_\kappa \mathsf{fact}\ 0\ (\mathsf{comp}\ \mathsf{id}\ ((*)\ 1)) \to \mathsf{comp}\ \mathsf{id}\ ((*)\ 1)\ 1 \to \mathsf{id}\ ((*)\ 1\ 1) \to_\kappa \mathsf{id}\ 1 \to 1$.

2.2 Accessibility and Computability

We recall the notion of computability with accessibility – which originates from [3] and is reformulated in [7] to couple with static dependency pairs – and adapt the notion of accessible function passing [7] to LCSTRSs.

Accessibility. Assume given a *sort ordering* – a quasi-ordering $\succsim$ over $\mathcal{S}$ whose strict part $\succ\, = \,\succsim \setminus \precsim$ is well-founded. We inductively define two relations $\succsim_+$ and $\succ_-$ over $\mathcal{S}$ and $\mathcal{T}$: given a sort A and a type $B = B_1 \to \cdots \to B_n \to C$ where C is a sort and $n \ge 0$, $A \succsim_+ B$ if and only if $A \succsim C$ and $\forall i.\, A \succ_- B_i$, and $A \succ_- B$ if and only if $A \succ C$ and $\forall i.\, A \succsim_+ B_i$.

Given a function symbol $f : A_1 \to \cdots \to A_n \to B$ where B is a sort, the set $\mathrm{Acc}(f)$ of the *accessible argument positions* of f is defined as $\{\, 1 \le i \le n \mid B \succsim_+ A_i \,\}$. A term t is called an *accessible subterm* of a term s, written as $s \trianglerighteq_{\mathrm{acc}} t$, if either $s = t$, or $s = f\ s_1 \cdots s_m$ for some $f \in \mathcal{F}$ and there exists $k \in \mathrm{Acc}(f)$ such that $s_k \trianglerighteq_{\mathrm{acc}} t$. An LCSTRS $\mathcal{R}$ is called *accessible function passing* (AFP) if there exists a sort ordering such that for all $f\ s_1 \cdots s_m \to r\ [\varphi] \in \mathcal{R}$ and $x \in \mathrm{Var}(f\ s_1 \cdots s_m) \cap \mathrm{Var}(r) \setminus \mathrm{Var}(\varphi)$, there exists k such that $s_k \trianglerighteq_{\mathrm{acc}} x$.

▶ **Example 3.** An LCSTRS $\mathcal{R}$ is AFP (with $\succsim$ equating all the sorts) if for all $f\ s_1 \cdots s_m \to r\ [\varphi] \in \mathcal{R}$ and $i \in \{\, 1, \ldots, m \,\}$, the type of each proper subterm of s_i is a sort. Rewrite rules for common higher-order functions, e.g., map and fold, usually fit this criterion.

Consider $\{\,\mathsf{complst}\ \mathsf{fnil}\ x \to x, \mathsf{complst}\ (\mathsf{fcons}\ f\ l)\ x \to \mathsf{complst}\ l\ (f\ x)\,\}$, where $\mathsf{complst} : \mathsf{funlist} \to \mathsf{int} \to \mathsf{int}$ composes a list of *functions*. This system is AFP with $\mathsf{funlist} \succ \mathsf{int}$.

The system $\{\,\mathsf{app}\ (\mathsf{lam}\ f) \to f\,\}$ in Section 1 is not AFP since $\mathsf{o} \succ \mathsf{o}$ cannot be true.

Computability. A term is called *neutral* if it takes the form $x\ t_1 \cdots t_n$ for some variable x. A set of *reducibility candidates*, or an *RC-set*, for the rewrite relation $\to_{\mathcal{R}}$ of an LCSTRS $\mathcal{R}$ is an $\mathcal{S}$-indexed family of sets $(I_A)_{A \in \mathcal{S}}$ (let I denote $\bigcup_A I_A$) satisfying the following conditions:

(1) Each element of I_A is a terminating (with respect to $\to_{\mathcal{R}}$) term of type A.

(2) Given terms s and t such that $s \to_{\mathcal{R}} t$, if s is in I_A, so is t.

(3) Given a neutral term s, if t is in I_A for all t such that $s \to_{\mathcal{R}} t$, so is s.

Given an RC-set I for $\to_{\mathcal{R}}$, a term t_0 is called *I-computable* if either the type of t_0 is a sort and $t_0 \in I$, or the type of t_0 is $A \to B$ and $t_0\ t_1$ is I-computable for all I-computable $t_1 : A$.

We are interested in a specific RC-set $\mathbb{C}$, whose existence is guaranteed by Theorem 4.

▶ **Theorem 4** (see [7]). *Given a sort ordering and an RC-set I for $\to_{\mathcal{R}}$, let $\Rrightarrow_I$ be the relation over terms such that $s \Rrightarrow_I t$ if and only if both s and t have a base type, $s = f\ s_1 \cdots s_m$ for some function symbol f, $t = s_k\ t_1 \cdots t_n$ for some $k \in \mathrm{Acc}(f)$ and t_i is I-computable for all i.*

Given an LCSTRS $\mathcal{R}$ with a sort ordering, there exists an RC-set $\mathbb{C}$ for $\to_{\mathcal{R}}$ such that $t \in \mathbb{C}_A$ if and only if $t : A$ is terminating with respect to $\to_{\mathcal{R}} \cup \Rrightarrow_{\mathbb{C}}$, and for all t' such that $t \to_{\mathcal{R}}^* t'$, if $t' = f\ t_1 \cdots t_n$ for some function symbol f, t_i is $\mathbb{C}$-computable for all $i \in \mathrm{Acc}(f)$.

Thus, given a $\mathbb{C}$-computable term $f\ t_1 \cdots t_n$, all its reducts and the accessible arguments – t_i for $i \in \mathrm{Acc}(f)$ – are also $\mathbb{C}$-computable. We consider $\mathbb{C}$-computability throughout this paper.

3 Static Dependency Pairs for LCSTRSs

Originally proposed for unconstrained first-order term rewriting, the dependency pair approach [1] – a methodology that analyzes the recursive structure of function calls – is at the heart of most modern automatic termination analyzers for various styles of term rewriting. There follow multiple higher-order generalizations, among which we adopt here the *static* branch [24, 7]. As we shall see in Section 5, this approach extends well to open-world analysis.

In this section, we adapt static dependency pairs to LCSTRSs. We start with a notation:

▶ **Definition 5.** *Given an LCSTRS $\mathcal{R}$, let $\mathcal{F}^\sharp$ be $\mathcal{F} \cup \{\, f^\sharp \mid f \in \mathcal{D}\,\}$ where $\mathcal{D}$ is the set of defined symbols in $\mathcal{R}$ and $f^\sharp$ is a fresh function symbol for all f. Let dp be a fresh sort, and for each defined symbol $f : A_1 \to \cdots \to A_n \to B$ where $B \in \mathcal{S}$, we assign $f^\sharp : A_1 \to \cdots \to A_n \to \mathsf{dp}$. Given a term $t = f\ t_1 \cdots t_n \in T(\mathcal{F}, \mathcal{V})$ where $f \in \mathcal{D}$, let $t^\sharp$ denote $f^\sharp\ t_1 \cdots t_n \in T(\mathcal{F}^\sharp, \mathcal{V})$.*

In the presence of logical constraints, a dependency pair should be more than a pair. Two extra components – a logical constraint and a set of variables – keep track of what substitutions are expected by the dependency pair.

▶ **Definition 6.** *A* static dependency pair *(SDP) is a quadruple $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L]$ where $s^\sharp$ and $t^\sharp$ are terms of type dp, φ is a logical constraint and $L \supseteq \mathrm{Var}(\varphi)$ is a set of variables whose types are theory sorts. Given a rewrite rule $\ell \to r\ [\varphi]$, let $\mathrm{SDP}(\ell \to r\ [\varphi])$ denote the set of SDPs of form $\ell^\sharp\ x_1 \cdots x_m \Rightarrow g^\sharp\ t_1 \cdots t_q\ y_{q+1} \cdots y_n\ [\varphi \mid \mathrm{Var}(\varphi) \cup (\mathrm{Var}(r) \setminus \mathrm{Var}(\ell))]$ such that*

(1) *$\ell^\sharp : A_1 \to \cdots \to A_m \to \mathsf{dp}$ while $x_i : A_i$ is a fresh variable for all i,*

(2) *$r\ x_1 \cdots x_m \trianglerighteq g\ t_1 \cdots t_q$ for $g \in \mathcal{D}$, and*

(3) *$g^\sharp : B_1 \to \cdots \to B_n \to \mathsf{dp}$ while $y_i : B_i$ is a fresh variable for all $i > q$.*

Let $\mathrm{SDP}(\mathcal{R})$ be $\bigcup_{\ell \to r\ [\varphi] \in \mathcal{R}} \mathrm{SDP}(\ell \to r\ [\varphi])$. A substitution σ is said to respect *an SDP $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L]$ if $\sigma(x)$ is a ground theory term for all $x \in L$ and $[\![\varphi\sigma]\!] = 1$.*

The component L is new compared to [15]. We shall see its usefulness in Section 4.4, as it gives us more freedom to manipulate dependency pairs. We introduce two shorthand notations for SDPs: $s^\sharp \Rightarrow t^\sharp\ [\varphi]$ for $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid \mathrm{Var}(\varphi)]$, and $s^\sharp \Rightarrow t^\sharp$ for $s^\sharp \Rightarrow t^\sharp\ [\mathsf{t} \mid \emptyset]$.

▶ **Example 7.** Consider the system $\mathcal{R}$ consisting of the following rewrite rules, in which $\mathsf{gcdlist} : \mathsf{intlist} \to \mathsf{int}$, $\mathsf{fold} : (\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{intlist} \to \mathsf{int}$ and $\mathsf{gcd} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$.

$$\begin{array}{llllll} \mathsf{gcdlist} \to \mathsf{fold}\ \mathsf{gcd}\ 0 & & \mathsf{fold}\ f\ y\ \mathsf{nil} \to y & & \mathsf{fold}\ f\ y\ (\mathsf{cons}\ x\ l) \to f\ x\ (\mathsf{fold}\ f\ y\ l) & \\ \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ (-m)\ n & [m < 0] & \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ m\ (-n) & [n < 0] & & \\ \mathsf{gcd}\ m\ 0 \to m & [m \geq 0] & \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ n\ (m \bmod n) & [m \geq 0 \wedge n > 0] & & \end{array}$$

The set $\mathrm{SDP}(\mathcal{R})$ consists of (1) $\mathsf{gcdlist}^\sharp\ l' \Rightarrow \mathsf{gcd}^\sharp\ m'\ n'$, (2) $\mathsf{gcdlist}^\sharp\ l' \Rightarrow \mathsf{fold}^\sharp\ \mathsf{gcd}\ 0\ l'$, (3) $\mathsf{fold}^\sharp\ f\ y\ (\mathsf{cons}\ x\ l) \Rightarrow \mathsf{fold}^\sharp\ f\ y\ l$, (4) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ (-m)\ n\ [m < 0]$, (5) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ m\ (-n)\ [n < 0]$, and (6) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$. Note that in (1), m' and n' occur on the right-hand side of $\Rightarrow$ but not on the left while they are *not* required to be instantiated to ground theory terms ($L = \emptyset$). This is normal for SDPs [7, 24].

Termination analysis via SDPs is based on the notion of a chain:

▶ **Definition 8.** *Given a set $\mathcal{P}$ of SDPs and a set $\mathcal{R}$ of rewrite rules, a* $(\mathcal{P}, \mathcal{R})$-chain *is a (finite or infinite) sequence $(s_0{}^\sharp \Rightarrow t_0{}^\sharp\ [\varphi_0 \mid L_0], \sigma_0), (s_1{}^\sharp \Rightarrow t_1{}^\sharp\ [\varphi_1 \mid L_1], \sigma_1), \ldots$ such that for all i, $s_i{}^\sharp \Rightarrow t_i{}^\sharp\ [\varphi_i \mid L_i] \in \mathcal{P}$, σ_i is a substitution which respects $s_i{}^\sharp \Rightarrow t_i{}^\sharp\ [\varphi_i \mid L_i]$, and $t_{i-1}{}^\sharp\sigma_{i-1} \to_{\mathcal{R}}^* s_i{}^\sharp\sigma_i$ if $i > 0$. The above $(\mathcal{P}, \mathcal{R})$-chain is called* computable *if $u\sigma_i$ is $\mathbb{C}$-computable for all i and u such that $t_i \rhd u$.*

▶ **Example 9.** Following Example 7, $(1, [l := \mathsf{nil}, m := 42, n := 24]), (6, [m := 42, n := 24]), (6, [m := 24, n := 18]), (6, [m := 18, n := 6])$ is a computable $(\mathrm{SDP}(\mathcal{R}), \mathcal{R})$-chain.

The key to establishing termination is the following result:

▶ **Theorem 10.** *An AFP system $\mathcal{R}$ is terminating if there exists no infinite computable* $(\mathrm{SDP}(\mathcal{R}), \mathcal{R})$*-chain.*

The proof (see [11, Appendix A.1]) is very similar to that for unconstrained SDPs [24, 7].

4 The Constrained DP Framework

In this section, we present several techniques based on SDPs, each as a class of *DP processors*; formally, we call this collection of DP processors the *constrained (static) DP framework*. In general, a DP framework [9, 7] constitutes a broad method for termination and non-termination. The presentation here is not complete – for example, we do not consider non-termination – and a complete one is beyond the scope of this paper. We rather focus on the most essential DP processors and those newly designed to handle logical constraints.

For presentation, we fix an LCSTRS $\mathcal{R}$.

▶ **Definition 11.** *A* DP problem *is a set $\mathcal{P}$ of SDPs. A DP problem $\mathcal{P}$ is called* finite *if there exists no infinite computable $(\mathcal{P}, \mathcal{R})$-chain. A* DP processor *is a partial mapping which possibly assigns to a DP problem a set of DP problems. A DP processor ρ is called* sound *if a DP problem $\mathcal{P}$ is finite whenever $\rho(\mathcal{P})$ consists only of finite DP problems.*

Following Theorem 10, in order to establish the termination of an AFP system $\mathcal{R}$, it suffices to show that $\mathrm{SDP}(\mathcal{R})$ is a finite DP problem. Given a collection of sound DP processors, we have the following procedure: (1) $Q := \{\,\mathrm{SDP}(\mathcal{R})\,\}$; (2) while Q contains a DP problem $\mathcal{P}$ to which some sound DP processor ρ is applicable, $Q := (Q \setminus \{\,\mathcal{P}\,\}) \cup \rho(\mathcal{P})$. If this procedure ends with $Q = \emptyset$, we can conclude that $\mathcal{R}$ is terminating.

4.1 The DP Graph and Its Approximations

The interconnection of SDPs via chains gives rise to a graph, namely, the DP graph [1], which models the reachability between dependency pairs. Since this graph is not computable in general, we follow the usual convention and consider its (over-)approximations:

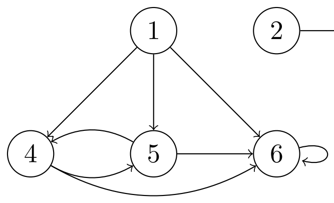
▶ **Definition 12.** *Given a set $\mathcal{P}$ of SDPs, a* graph approximation *(G_θ, θ) for $\mathcal{P}$ consists of a finite directed graph G_θ and a mapping θ which assigns to each SDP in $\mathcal{P}$ a vertex of G_θ so that there is an edge from $\theta(p_0)$ to $\theta(p_1)$ whenever $(p_0, \sigma_0), (p_1, \sigma_1)$ is a $(\mathcal{P}, \mathcal{R})$-chain for some substitutions σ_0 and σ_1.*

Here (G_θ, θ) approximates the true DP graph by allowing θ to assign a single vertex to multiple (possibly, infinitely many) SDPs, and by allowing G_θ to contain an edge from $\theta(p_0)$ to $\theta(p_1)$ even if p_0 and p_1 are not connected by any $(\mathcal{P}, \mathcal{R})$-chain. In practice, we typically deal with only a finite set $\mathcal{P}$ of SDPs, in which case we usually take a bijection for θ.

This graph structure is useful because we can leverage it to decompose the DP problem.

▶ **Definition 13.** *Given a DP problem $\mathcal{P}$, a* graph processor *computes a graph approximation (G_θ, θ) for $\mathcal{P}$ and the strongly connected components (SCCs) of G_θ, then returns $\{\, \{\, p \in \mathcal{P} \mid \theta(p)$ belongs to $S \,\} \mid S$ is a non-trivial SCC of $G_\theta \,\}$.*

▶ **Example 14.** Following Example 7, a (tight) graph approximation for $\mathrm{SDP}(\mathcal{R})$ is in Figure 1. If a graph processor produces this graph as the graph approximation, it will return the set of DP problems $\{\, \{\, 3 \,\}, \{\, 4, 5 \,\}, \{\, 6 \,\} \,\}$.



$$\begin{tikzcd}
 & 1 \arrow[dl] \arrow[d] \arrow[dr] & 2 \arrow[r] & 3 \arrow[loop right] \\
4 \arrow[r, bend right] \arrow[rr, bend right] & 5 \arrow[l, bend right] \arrow[r] & 6 \arrow[loop right] &
\end{tikzcd}$$

Figure 1 A graph approximation for $\mathrm{SDP}(\mathcal{R})$ from Example 7.

Implementation. To compute a graph approximation, we adapt the common CAP approach [10, 33] and take theories into account. Considering theories allows us, for example, *not* to have an edge from (6) to (4) in Figure 1.

We assume given a finite set of SDPs and let θ be a bijection. Whether there is an edge from $\theta({s_0}^\sharp \Rightarrow {t_0}^\sharp\ [\varphi_0 \mid L_0])$ to $\theta({s_1}^\sharp \Rightarrow {t_1}^\sharp\ [\varphi_1 \mid L_1])$ – we rename variables if necessary to avoid name collisions between the two SDPs – is determined by the satisfiability (which we check by an SMT solver) of $\varphi_0 \wedge \varphi_1 \wedge \zeta({t_0}^\sharp, {s_1}^\sharp)$ where $\zeta(u, v)$ is defined as follows:

- If $u = f\ u_1 \cdots u_n$ where $f \in \mathcal{F}^\sharp$ and no rewrite rule in $\mathcal{R}$ takes the form $f\ \ell_1 \cdots \ell_k \to r\ [\varphi]$ for $k \leq n$, we define $\zeta(u, v)$ in two cases:
  (1) $\zeta(u, v) = \zeta(u_1, v_1) \wedge \cdots \wedge \zeta(u_n, v_n)$ if $v = f\ v_1 \cdots v_n$.
  (2) $\zeta(u, v) = \mathfrak{f}$ if $v = g\ v_1 \cdots v_m$ for some function symbol g other than f, and either f is not a theory symbol or g is not a value.
- Suppose $\zeta(u, v)$ is not defined above; $\zeta(u, v) = (u \equiv v)$ if $u \in T(\mathcal{F}_\vartheta, L_0)$ has a base type and v is a theory term in which the type of each variable is a theory sort, and $\zeta(u, v) = \mathfrak{t}$ otherwise.

See [11, Appendix A.2] for the proof that this approach produces a graph approximation.

Then strongly connected components can be computed by Tarjan's algorithm [32].

▶ **Example 15.** In Figure 1, since $(m_0 \geq 0 \wedge n_0 > 0) \wedge m_1 < 0 \wedge (n_0 \equiv m_1 \wedge m_0 \bmod n_0 \equiv n_1)$ is unsatisfiable, there is no edge from (6) to (4).

4.2 The Subterm Criterion

The subterm criterion [13, 24] handles structural recursion and allows us to remove decreasing SDPs without considering rewrite rules in $\mathcal{R}$. We start with defining projections:

▶ **Definition 16.** *Let* $\mathrm{heads}(\mathcal{P})$ *denote the set of function symbols heading either side of an SDP in $\mathcal{P}$. A* projection ν *for a set $\mathcal{P}$ of SDPs is a mapping from* $\mathrm{heads}(\mathcal{P})$ *to integers such that $1 \leq \nu(f^\sharp) \leq n$ if $f^\sharp : A_1 \to \cdots \to A_n \to \mathsf{dp}$. Let $\bar{\nu}(f^\sharp\ t_1 \cdots t_n)$ denote $t_{\nu(f^\sharp)}$.*

A projection chooses an argument position for each relevant function symbol so that arguments at those positions do not increase in a chain.

▶ **Definition 17.** *Given a set $\mathcal{P}$ of SDPs, a projection ν is said to* $\rhd$-orient *a subset $\mathcal{P}'$ of $\mathcal{P}$ if $\bar{\nu}(s^\sharp) \rhd \bar{\nu}(t^\sharp)$ for all $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L] \in \mathcal{P}'$ and $\bar{\nu}(s^\sharp) = \bar{\nu}(t^\sharp)$ for all $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L] \in \mathcal{P} \setminus \mathcal{P}'$. A* subterm criterion processor *assigns to a DP problem $\mathcal{P}$ the singleton $\{\, \mathcal{P} \setminus \mathcal{P}' \,\}$ for some non-empty subset $\mathcal{P}'$ of $\mathcal{P}$ such that there exists a projection for $\mathcal{P}$ which $\rhd$-orients $\mathcal{P}'$.*

▶ **Example 18.** Following Example 14, a subterm criterion processor is applicable to $\{\, 3 \,\}$. Let $\nu(\mathsf{fold}^\sharp)$ be 3 so that $\bar{\nu}(\mathsf{fold}^\sharp\ f\ y\ (\mathsf{cons}\ x\ l)) = \mathsf{cons}\ x\ l \rhd l = \bar{\nu}(\mathsf{fold}^\sharp\ f\ y\ l)$. The processor returns $\{\, \emptyset \,\}$, and the empty DP problem can (trivially) be removed by a graph processor.

Implementation. The search for a suitable projection can be done through SMT and is standard: we introduce an integer variable $N_{f^\sharp}$ that represents $\nu(f^\sharp)$ for each $f^\sharp \in \mathrm{heads}(\mathcal{P})$, and a boolean variable strict_p for each $p \in \mathcal{P}$; then we encode the requirement per SDP.

4.3 Integer Mappings

The subterm criterion deals with recursion over the structure of terms, but not recursion over, say, integers, which requires us to utilize the information in logical constraints. In this subsection, we assume that $\mathsf{int} \in \mathcal{S}_\vartheta$ and $\mathcal{F}_\vartheta \supseteq \{\, \geq, >, \wedge \,\}$, where $\geq : \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$, $> : \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$ and $\wedge : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$ are interpreted in the standard way.

▶ **Definition 19.** *Given a set $\mathcal{P}$ of SDPs, for all $f^\sharp \in$* $\mathrm{heads}(\mathcal{P})$ *(see Definition 16) where $f^\sharp : A_1 \to \cdots \to A_n \to \mathsf{dp}$, let $\iota(f^\sharp)$ be the subset of $\{\, 1, \ldots, n \,\}$ such that $i \in \iota(f^\sharp)$ if and only if $A_i \in \mathcal{S}_\vartheta$ and the i-th argument of any occurrence of $f^\sharp$ in an SDP $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L] \in \mathcal{P}$ is in $T(\mathcal{F}_\vartheta, L)$. Let $\mathcal{X}(f^\sharp)$ be a set of fresh variables $\{\, x_{f^\sharp,i} \mid i \in \iota(f^\sharp) \,\}$ where $x_{f^\sharp,i} : A_i$ for all i. An* integer mapping $\mathcal{J}$ *for $\mathcal{P}$ is a mapping from* $\mathrm{heads}(\mathcal{P})$ *to theory terms such that for all $f^\sharp$, $\mathcal{J}(f^\sharp) : \mathsf{int}$ and* $\mathrm{Var}(\mathcal{J}(f^\sharp)) \subseteq \mathcal{X}(f^\sharp)$. *Let $\bar{\mathcal{J}}(f^\sharp\ t_1 \cdots t_n)$ denote $\mathcal{J}(f^\sharp)[x_{f^\sharp,i} := t_i]_{i \in \iota(f^\sharp)}$.*

With integer mappings, we can handle decreasing integer values.

▶ **Definition 20.** *Given a set $\mathcal{P}$ of SDPs, an integer mapping $\mathcal{J}$ is said to* $>$-orient *a subset $\mathcal{P}'$ of $\mathcal{P}$ if $\varphi \models \bar{\mathcal{J}}(s^\sharp) \geq 0 \wedge \bar{\mathcal{J}}(s^\sharp) > \bar{\mathcal{J}}(t^\sharp)$ for all $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L] \in \mathcal{P}'$, and $\varphi \models \bar{\mathcal{J}}(s^\sharp) \geq \bar{\mathcal{J}}(t^\sharp)$ for all $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L] \in \mathcal{P} \setminus \mathcal{P}'$, where $\varphi \models \varphi'$ denotes that $[\![\varphi\sigma]\!] = 1$ implies $[\![\varphi'\sigma]\!] = 1$ for each substitution σ which maps variables in* $\mathrm{Var}(\varphi) \cup \mathrm{Var}(\varphi')$ *to values. An* integer mapping processor *assigns to a DP problem $\mathcal{P}$ the singleton $\{\, \mathcal{P} \setminus \mathcal{P}' \,\}$ for some non-empty subset $\mathcal{P}'$ of $\mathcal{P}$ such that there exists an integer mapping for $\mathcal{P}$ which $>$-orients $\mathcal{P}'$.*

▶ **Example 21.** Following Example 14, an integer mapping processor is applicable to $\{6\}$. Let $\mathcal{J}(\mathsf{gcd}^\sharp)$ be $x_{\mathsf{gcd}^\sharp,2}$ so that $\bar{\mathcal{J}}(\mathsf{gcd}^\sharp\ m\ n) = n$, $\bar{\mathcal{J}}(\mathsf{gcd}^\sharp\ n\ (m \bmod n)) = m \bmod n$ and $m \geq 0 \wedge n > 0 \models n \geq 0 \wedge n > m \bmod n$. The processor returns $\{\emptyset\}$, and the empty DP problem can (trivially) be removed by a graph processor.

Implementation. There are several ways to implement integer mapping processors. In our implementation, we generate a number of "interpretation candidates" and use an SMT encoding to select for each $f^\sharp \in \mathrm{heads}(\mathcal{P})$ one candidate that satisfies the requirements. Candidates include forms such as $\mathcal{J}(f^\sharp) = x_{f^\sharp,i}$ and those that are generated from the SDPs' logical constraints – e.g., given $\mathsf{f}^\sharp\ x\ y \Rightarrow \mathsf{g}^\sharp\ x\ (y+1)\ [y < x]$, we generate $\mathcal{J}(\mathsf{f}^\sharp) = x_{\mathsf{f}^\sharp,1} - x_{\mathsf{f}^\sharp,2} - 1$ because $y < x$ implies $x - y - 1 \geq 0$.

4.4 Theory Arguments

Integer mapping processors have a clear limitation: what if some key variables do not occur in the set L? This is observed in the remaining DP problem $\{4, 5\}$ from Example 7. It is clearly finite but no integer mapping processor is applicable since $\iota(\mathsf{gcd}^\sharp) = \emptyset$.

This restriction exists for a reason. Variables that are not guaranteed to be instantiated to theory terms may be instantiated to *non-deterministic* terms – e.g., $\{\mathsf{f}^\sharp\ x\ y\ z \Rightarrow \mathsf{f}^\sharp\ x\ (x+1)\ (x-1)\ [y < z]\}$ is not a finite DP problem if $\mathcal{R} \supseteq \{\mathsf{c}\ x\ y \to x, \mathsf{c}\ x\ y \to y\}$.

The problem of $\{4, 5\}$ arises because each SDP focuses on only one argument: for example, the logical constraint (with the component L) of (5) only concerns n so in principle we cannot assume anything about m. Yet, if (5) follows (4) in a chain, we *can* derive that m must be instantiated to a ground theory term (we call such an argument a *theory argument*). We explore a way of propagating this information.

▶ **Definition 22.** *A* theory argument (position) mapping *τ for a set $\mathcal{P}$ of SDPs is a mapping from* heads$(\mathcal{P})$ *(see Definition 16) to subsets of $\mathbb{Z}$ such that $\tau(f^\sharp) \subseteq \{1 \leq i \leq m \mid A_i \in \mathcal{S}_\vartheta\}$ if $f^\sharp : A_1 \to \cdots \to A_m \to \mathsf{dp}$, s_i is a theory term and the type of each variable in* $\mathrm{Var}(s_i)$ *is a theory sort for all $f^\sharp\ s_1 \cdots s_m \Rightarrow t^\sharp\ [\varphi \mid L] \in \mathcal{P}$ and $i \in \tau(f^\sharp)$, and t_j is a theory term and* $\mathrm{Var}(t_j) \subseteq L \cup \bigcup_{i\in\tau(f^\sharp)} \mathrm{Var}(s_i)$ *for all $f^\sharp\ s_1 \cdots s_m \Rightarrow g^\sharp\ t_1 \cdots t_n\ [\varphi \mid L] \in \mathcal{P}$ and $j \in \tau(g^\sharp)$. Let $\bar{\tau}(f^\sharp\ s_1 \cdots s_m \Rightarrow t^\sharp\ [\varphi \mid L])$ denote $f^\sharp\ s_1 \cdots s_m \Rightarrow t^\sharp\ [\varphi \mid L \cup \bigcup_{i\in\tau(f^\sharp)} \mathrm{Var}(s_i)]$.*

By a theory argument mapping, we choose a subset of the given set of SDPs from which the theory argument information is propagated.

▶ **Definition 23.** *Given a set $\mathcal{P}$ of SDPs, a theory argument mapping τ is said to* fix *a subset $\mathcal{P}'$ of $\mathcal{P}$ if $\bigcup_{i\in\tau(f^\sharp)} \mathrm{Var}(t_i) \subseteq L$ for all $s^\sharp \Rightarrow f^\sharp\ t_1 \cdots t_n\ [\varphi \mid L] \in \mathcal{P}'$. A* theory argument processor *assigns to a DP problem $\mathcal{P}$ the pair $\{\{\bar{\tau}(p) \mid p \in \mathcal{P}\}, \mathcal{P} \setminus \mathcal{P}'\}$ for some non-empty subset $\mathcal{P}'$ of $\mathcal{P}$ such that there exists a theory argument mapping for $\mathcal{P}$ which fixes $\mathcal{P}'$.*

▶ **Example 24.** Following Example 14, a theory argument processor is applicable to $\{4, 5\}$. Let $\tau(\mathsf{gcd}^\sharp)$ be $\{1\}$ so that τ fixes $\{4\}$. The processor returns the pair $\{\{4, (7)\ \mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ m\ (-n)\ [n < 0 \mid \{m, n\}]\}, \{5\}\}$. The integer mapping processor with $\mathcal{J}(\mathsf{gcd}^\sharp) = -x_{\mathsf{gcd}^\sharp,1}$ removes (4) from $\{4, 7\}$. Then $\{7\}$ and $\{5\}$ can be removed by graph processors.

Implementation. To find a valid theory argument mapping, we simply start by setting $\tau(f^\sharp) = \{1, \ldots, m\}$ for all $f^\sharp$, and choose one SDP to fix. Then we iteratively remove arguments that do not satisfy the condition until no such argument is left.

4.5 Reduction Pairs

Although it is not needed by the running example, we present a constrained variant of *reduction pair processors*, which are at the heart of most unconstrained termination analyzers.

▶ **Definition 25.** *A* constrained relation *R is a set of quadruples (s, t, φ, L) where s and t are terms which have the same type, φ is a logical constraint and $L \supseteq \mathrm{Var}(\varphi)$ is a set of variables whose types are theory sorts. We write $s\ R^L_\varphi\ t$ if $(s, t, \varphi, L) \in R$. A binary relation R' over terms is said to* cover *a constrained relation R if $s\ R^L_\varphi\ t$ implies that $(s\sigma)\downarrow_\kappa R'\ (t\sigma)\downarrow_\kappa$ for each substitution σ such that $\sigma(x)$ is a ground theory term for all $x \in L$ and $[\![\varphi\sigma]\!] = 1$.*

A constrained reduction pair *$(\succeq, \succ)$ is a pair of constrained relations such that $\succeq$ is covered by some reflexive relation $\sqsupseteq$ which includes $\to_\kappa$ and is monotonic (i.e., $s \sqsupseteq t$ implies $C[s] \sqsupseteq C[t]$), $\succ$ is covered by some well-founded relation $\sqsupset$, and $\sqsupseteq ; \sqsupset \subseteq \sqsupset^+$.*

▶ **Definition 26.** *A* reduction pair processor *assigns to a DP problem $\mathcal{P}$ the singleton $\{\, \mathcal{P} \backslash \mathcal{P}' \,\}$ for some non-empty subset $\mathcal{P}'$ of $\mathcal{P}$ such that there exists a constrained reduction pair $(\succeq, \succ)$ where (1) $s^\sharp \succ^L_\varphi t^\sharp$ for all $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L] \in \mathcal{P}'$, (2) $s^\sharp \succeq^L_\varphi t^\sharp$ for all $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L] \in \mathcal{P} \setminus \mathcal{P}'$, and (3) $\ell \succeq_\varphi^{\mathrm{Var}(\varphi) \cup (\mathrm{Var}(r) \setminus \mathrm{Var}(\ell))} r$ for all $\ell \to r\ [\varphi] \in \mathcal{R}$.*

While a variety of reduction pairs have been proposed for unconstrained rewriting, it is not yet the case in a higher-order and constrained setting: so far the only one is a limited version of HORPO [12], which is adapted into a weakly monotonic reduction pair [18] and then implemented in the DP framework. This is still a prototype definition.

We have included reduction pair processors here because their definition allows us to start designing constrained reduction pairs. In particular, as unconstrained reduction pairs can be used as the covering pair $(\sqsupseteq, \sqsupset)$, it is likely that many of them (such as variants of HORPO and weakly monotonic algebras) can be adapted.

We conclude this section by the following result (see [11, Appendix A.2]):

▶ **Theorem 27.** *All the DP processors defined in Section 4 are sound.*

5 Universal Computability

Termination is not a *modular* property: given terminating systems $\mathcal{R}_0$ and $\mathcal{R}_1$, we cannot generally conclude that $\mathcal{R}_0 \cup \mathcal{R}_1$ is also terminating. As computability is based on termination, it is not modular either. For example, both $\{\, \mathsf{a} \to \mathsf{b} \,\}$ and $\{\, \mathsf{f}\ \mathsf{b} \to \mathsf{f}\ \mathsf{a} \,\}$ are terminating, and $\mathsf{f} : \mathsf{o} \to \mathsf{o}$ is computable in the second system; yet, combining the two yields $\mathsf{f}\ \mathsf{a} \to \mathsf{f}\ \mathsf{b} \to \mathsf{f}\ \mathsf{a} \to \cdots$, which refutes the termination of the combination and the computability of f.

On the other hand, functions like map and fold are prevalently used; the lack of a modular principle for the termination analysis of higher-order systems involving such functions is painful. Moreover, if such a system is non-terminating, this is seldom attributed to those functions, which are generally considered "terminating" regardless of how they may be called.

In this section, we propose *universal computability*, a concept which corresponds to the termination of a function in all "reasonable" uses. First, we rephrase the notion of a hierarchical combination [27, 28, 29, 6] in terms of LCSTRSs:

▶ **Definition 28.** *An LCSTRS $\mathcal{R}_1$ is called an* extension *of a base system $\mathcal{R}_0$ if the two systems' interpretations of theory symbols coincide over all the theory symbols in common, and function symbols in $\mathcal{R}_0$ are not defined by any rewrite rule in $\mathcal{R}_1$. Given a base system $\mathcal{R}_0$ and an extension $\mathcal{R}_1$ of $\mathcal{R}_0$, the system $\mathcal{R}_0 \cup \mathcal{R}_1$ is called a* hierarchical combination.

In a hierarchical combination, function symbols in the base system can occur in the extension, but cannot be (re)defined. This forms the basis of the modular programming scenario we are interested in: think of the base system as a library containing the definitions of, say, map and fold. We further define a class of extensions to take information hiding into account:

▶ **Definition 29.** *Given an LCSTRS $\mathcal{R}_0$ and a set of function symbols – called* hidden symbols *– in $\mathcal{R}_0$, an extension $\mathcal{R}_1$ of $\mathcal{R}_0$ is called a* public extension *if hidden symbols do not occur in any rewrite rule in $\mathcal{R}_1$.*

Now we present the central definitions of this section:

▶ **Definition 30.** *Given an LCSTRS $\mathcal{R}_0$ with a sort ordering $\succsim$, a term t is called* universally computable *if for each extension $\mathcal{R}_1$ of $\mathcal{R}_0$ and each extension $\succsim'$ of $\succsim$ to sorts in $\mathcal{R}_0 \cup \mathcal{R}_1$ (i.e., $\succsim'$ coincides with $\succsim$ over sorts in $\mathcal{R}_0$), t is $\mathbb{C}$-computable in $\mathcal{R}_0 \cup \mathcal{R}_1$ with $\succsim'$; if a set of hidden symbols in $\mathcal{R}_0$ is also given and the above universal quantification of $\mathcal{R}_1$ is restricted to* public *extensions, such a term t is called* publicly computable*.*

The base system $\mathcal{R}_0$ is called universally computable if all its terms are; it is called publicly computable if all its public *terms – terms that contain no hidden symbol – are.*

With an empty set of hidden symbols, the two notions – universal computability and public computability – coincide. Below we state common properties in terms of public computability.

In summary, we consider passing $\mathbb{C}$-computable arguments to a defined symbol in $\mathcal{R}_0$ the "reasonable" way of calling the function. To establish the universal computability of higher-order functions such as map and fold – i.e., to prove that they are $\mathbb{C}$-computable in *all* relevant hierarchical combinations – we will use SDPs, which are about $\mathbb{C}$-computability.

▶ **Example 31.** The system $\{ \mathsf{app}\ (\mathsf{lam}\ f) \to f \}$ in Section 1 is not universally computable due to the extension $\{ \mathsf{w}\ x \to \mathsf{app}\ x\ x \}$.

5.1 The DP Framework Revisited

To use SDPs for universal – or public – computability, we need a more general version of Theorem 10. We start with defining public chains:

▶ **Definition 32.** *An SDP $f^\sharp\ s_1 \cdots s_m \Rightarrow t^\sharp\ [\varphi \mid L]$ is called public if f is not a hidden symbol. A $(\mathcal{P}, \mathcal{R})$-chain is called public if its first SDP is public.*

Now we state the main result of this section:

▶ **Theorem 33.** *An AFP system $\mathcal{R}_0$ with sort ordering $\succsim$ is publicly computable with respect to a set of hidden symbols in $\mathcal{R}_0$ if there exists no infinite computable $(\mathrm{SDP}(\mathcal{R}_0), \mathcal{R}_0 \cup \mathcal{R}_1)$-chain that is* public *for each public extension $\mathcal{R}_1$ of $\mathcal{R}_0$ and each extension $\succsim'$ of $\succsim$ to sorts in $\mathcal{R}_0 \cup \mathcal{R}_1$.*

While this result is not surprising and its proof (see [11, Appendix A.3]) is standard, it is not obvious how it can be used. The key observation which enables us to use the DP framework for public computability is that among the DP processors in Section 4, only reduction pair processors rely on the rewrite rules of the underlying system $\mathcal{R}$ (depending on how it computes a graph approximation, a graph processor does not have to know all the rewrite rules). Henceforth, we fix a base system $\mathcal{R}_0$, a set of hidden symbols in $\mathcal{R}_0$ and an arbitrary, unknown public extension $\mathcal{R}_1$ of $\mathcal{R}_0$. Now $\mathcal{R}$ is the hierarchical combination $\mathcal{R}_0 \cup \mathcal{R}_1$.

First, we generalize the definition of a DP problem:

▶ **Definition 34.** *A* (universal) DP problem $(\mathcal{P}, \mathtt{p})$ *consists of a set $\mathcal{P}$ of SDPs and a flag* $\mathtt{p} \in \{\, \mathfrak{an}, \mathfrak{pu} \,\}$ *(for* any *or* public*). A DP problem $(\mathcal{P}, \mathtt{p})$ is called* finite *if either (1)* $\mathtt{p} = \mathfrak{an}$ *and there exists no infinite computable $(\mathcal{P}, \mathcal{R}_0 \cup \mathcal{R}_1)$-chain, or (2)* $\mathtt{p} = \mathfrak{pu}$ *and there exists no infinite computable $(\mathcal{P}, \mathcal{R}_0 \cup \mathcal{R}_1)$-chain which is* public*.*

DP processors are defined in the same way as before, now for universal DP problems. The goal is to show that $(\mathrm{SDP}(\mathcal{R}_0), \mathfrak{pu})$ is finite, and the procedure for termination in Section 4 also works here if we change the initialization of Q accordingly.

Next, we review the DP processors presented in Section 4. For each ρ of the original graph, subterm criterion and integer mapping processors, the processor ρ' such that $\rho'(\mathcal{P}, \mathtt{p}) = \{\, (\mathcal{P}', \mathfrak{an}) \mid \mathcal{P}' \in \rho(\mathcal{P}) \,\}$ is sound for universal DP problems. For theory argument processors, we can do better when the input flag is $\mathfrak{pu}$ (when it is $\mathfrak{an}$, we just handle $\mathcal{P}$ in the same way as we do in Section 4 and the output flags are obviously $\mathfrak{an}$): if the subset $\mathcal{P}'$ of $\mathcal{P}$ fixed by a theory argument mapping τ contains all the public SDPs in $\mathcal{P}$, the processor should return the singleton $\{\, (\{\, p \mid p \in \mathcal{P} \text{ is public} \,\} \cup \{\, \bar{\tau}(p) \mid p \in \mathcal{P} \text{ is not public} \,\}, \mathfrak{pu}) \,\}$; otherwise, the pair $\{\, (\{\, \bar{\tau}(p) \mid p \in \mathcal{P} \,\}, \mathfrak{an}), (\mathcal{P} \setminus \mathcal{P}', \mathfrak{pu}) \,\}$. Reduction pair processors require knowledge of the extension $\mathcal{R}_1$ so we do not adapt them.

New Processors. Last, we propose two classes of DP processors that are useful for public computability. Processors of the first class do not actually simplify DP problems; they rather alter their input to allow other DP processors to be applied subsequently.

▶ **Definition 35.** *Given sets $\mathcal{P}_1$ and $\mathcal{P}_2$ of SDPs, $\mathcal{P}_2$ is said to* cover *$\mathcal{P}_1$ if for each SDP $s^\sharp \Rightarrow t^\sharp\ [\varphi_1 \mid L_1] \in \mathcal{P}_1$ and each substitution σ_1 which respects $s^\sharp \Rightarrow t^\sharp\ [\varphi_1 \mid L_1]$, there exist an SDP $s^\sharp \Rightarrow t^\sharp\ [\varphi_2 \mid L_2] \in \mathcal{P}_2$ and a substitution σ_2 such that σ_2 respects $s^\sharp \Rightarrow t^\sharp\ [\varphi_2 \mid L_2]$, $s\sigma_1 = s\sigma_2$ and $t\sigma_1 = t\sigma_2$. A* constraint modification processor *assigns to a DP problem $(\mathcal{P}, \mathtt{p})$ the singleton $\{\, (\mathcal{P}', \mathtt{p}) \,\}$ for some $\mathcal{P}'$ which covers $\mathcal{P}$.*

Now combined with the information of hidden symbols, the DP graph allows us to remove SDPs that are unreachable from any public SDP.

▶ **Definition 36.** *A* reachability processor *assigns to a DP problem $(\mathcal{P}, \mathfrak{pu})$ the singleton $\{\, (\{\, p \in \mathcal{P} \mid \theta(p)$ is reachable from $\theta(p_0)$ for some public SDP $p_0 \,\}, \mathfrak{pu}) \,\}$, given a graph approximation (G_θ, θ) for $\mathcal{P}$.*

These two classes of DP processors are often used together: a constraint modification processor can split an SDP into simpler ones, some of which may be removed by a reachability processor. In our implementation, a constraint modification processor is particularly used to break an SDP $s^\sharp \Rightarrow t^\sharp\ [u \neq v \mid L]$ into two SDPs with logical constraints $u < v$ and $u > v$, respectively (see Example 37); similarly for $s^\sharp \Rightarrow t^\sharp\ [u \vee v \mid L]$.

▶ **Example 37.** Consider an alternative implementation of the factorial function from Example 2, which has SDPs (1) $\mathsf{fact}^\sharp\ n\ k \Rightarrow \mathsf{comp}^\sharp\ k\ ((*)\ n)\ x'\ [n \neq 0]$, (2) $\mathsf{fact}^\sharp\ n\ k \Rightarrow \mathsf{fact}^\sharp\ (n-1)\ (\mathsf{comp}\ k\ ((*)\ n))\ [n \neq 0]$, and (3) $\mathsf{init}^\sharp\ k \Rightarrow \mathsf{fact}^\sharp\ 42\ k$. Assume that fact is a hidden symbol. Note that $(\{\, 1, 2, 3 \,\}, \mathfrak{pu})$ is not finite without this assumption. A constraint modification processor can replace (2) with (4) $\mathsf{fact}^\sharp\ n\ k \Rightarrow \mathsf{fact}^\sharp\ (n-1)\ (\mathsf{comp}\ k\ ((*)\ n))\ [n < 0]$, and (5) $\mathsf{fact}^\sharp\ n\ k \Rightarrow \mathsf{fact}^\sharp\ (n-1)\ (\mathsf{comp}\ k\ ((*)\ n))\ [n > 0]$. A reachability processor can then remove (4). The remaining DP problem $(\{\, 1, 3, 5 \,\}, \mathfrak{pu})$ can easily be handled by a graph processor and an integer mapping processor.

We conclude this section by the following result (see [11, Appendix A.4]):

▶ **Theorem 38.** *All the DP processors defined in Section 5 are sound.*

6 Experiments and Future Work

All the results in this paper have been implemented in our open-source analyzer Cora [20]. We have evaluated Cora on three groups of experiments, and the results are in Table 1.

Table 1 Cora experiment results.

	Custom	STRS	ITRS
Termination	20/28	72/140	69/117
Computability	20/28	66/140	68/117
Wanda	–	105/140	–
AProVE	–	–	102/117

The first group contains examples in this paper and several other LC(S)TRS benchmarks we have collected. The second group contains all the λ-free problems from the higher-order category of TPDB [5]. The third group contains problems from the first-order "integer TRS innermost" category. The computability tests analyze public computability; since there are no hidden symbols in TPDB, the main difference from a termination check is that reduction pair processors are disabled. A full evaluation page is available through the link:

`https://www.cs.ru.nl/~cynthiakop/experiments/mfcs2024`

Unsurprisingly, Cora is substantially weaker than Wanda [16] on unconstrained higher-order TRSs, and AProVE [8] on first-order integer TRSs: this work aims to be a starting point for *combining* higher-order term analysis and theory reasoning, and cannot yet compete with dedicated tools that have had years of development. Nevertheless, we believe that these results show a solid foundation with only a handful of simple techniques.

Future Work. Many of the existing techniques used in the analyses of integer TRSs and higher-order TRSs are likely to be extensible to our setting, leaving many encouraging avenues for further development. We highlight the most important few:

- Usable rules with respect to an argument filtering [10, 17]. To effectively use reduction pairs, being able to discard some rewrite rules is essential (especially for universal computability, if we can discard the unknown ones). Closely related is the adaptation of more reduction pairs such as weakly monotonic algebras [36, 34], tuple interpretations [22, 35] and more sophisticated path orderings [4], all of which have higher-order formulations.
- Transformation techniques, such as narrowing, and chaining dependency pairs together (as used for instance for integer transition systems [8, Secion 3.1]). This could also be a step toward using the constrained DP framework for non-termination.
- Handling the innermost or call-by-value strategy. Several functional languages adopt call-by-value evaluation, and applying this restriction may allow for more powerful analyses. In the first-order DP framework, there is ample work on the innermost strategy to build on (see, e.g., [9, 10]).
- Theory-specific processors for popular theories other than integers, e.g., bit vectors [25].

References

1 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236(1–2):133–178, 2000. doi:10.1016/S0304-3975(99)00207-8.

2 F. Blanqui. Higher-order dependency pairs. In A. Geser and H. Søndergaard, editors, *Proc. WST*, pages 22–26, 2006. doi:10.48550/arXiv.1804.08855.

3 F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive-data-type systems. *TCS*, 272(1–2):41–68, 2002. doi:10.1016/S0304-3975(00)00347-9.

4 F. Blanqui, J.-P. Jouannaud, and A. Rubio. The computability path ordering: the end of a quest. In M. Kaminski and S. Martini, editors, *Proc. CSL*, pages 1–14, 2008. doi:10.1007/978-3-540-87531-4_1.

5 Community. The termination problem database (TPDB). URL: https://github.com/TermCOMP/TPDB.

6 N. Dershowitz. Hierarchical termination. In N. Dershowitz and N. Lindenstrauss, editors, *Proc. CTRS*, pages 89–105, 1995. doi:10.1007/3-540-60381-6_6.

7 C. Fuhs and C. Kop. A static higher-order dependency pair framework. In L. Caires, editor, *Proc. ESOP*, pages 752–782, 2019. doi:10.1007/978-3-030-17184-1_27.

8 J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *JAR*, 58:3–31, 2017. doi:10.1007/s10817-016-9388-y.

9 J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: combining techniques for automated termination proofs. In F. Baader and A. Voronkov, editors, *Proc. LPAR*, pages 301–331, 2005. doi:10.1007/978-3-540-32275-7_21.

10 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *JAR*, 37:155–203, 2006. doi:10.1007/s10817-006-9057-7.

11 L. Guo, K. Hagens, C. Kop, and D. Vale. Higher-order constrained dependency pairs for (universal) computability. Technical report, Radboud University, 2024. doi:10.48550/arXiv.2406.19379.

12 L. Guo and C. Kop. Higher-order LCTRSs and their termination. In S. Weirich, editor, *Proc. ESOP*, pages 331–357, 2024. doi:10.1007/978-3-031-57267-8_13.

13 N. Hirokawa and A. Middeldorp. Dependency pairs revisited. In V. van Oostrom, editor, *Proc. RTA*, pages 249–268, 2004. doi:10.1007/978-3-540-25979-4_18.

14 J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In G. Longo, editor, *Proc. LICS*, pages 402–411, 1999. doi:10.1109/LICS.1999.782635.

15 C. Kop. Termination of LCTRSs. In J. Waldmann, editor, *Proc. WST*, pages 59–63, 2013. doi:10.48550/arXiv.1601.03206.

16 C. Kop. WANDA – A higher order termination tool. In Z. M. Ariola, editor, *Proc. FSCD*, pages 36:1–36:19, 2020. doi:10.4230/LIPIcs.FSCD.2020.36.

17 C. Kop. Cutting a proof into bite-sized chunks: incrementally proving termination in higher-order term rewriting. In A. P. Felty, editor, *Proc. FSCD*, pages 1:1–1:17, 2022. doi:10.4230/LIPIcs.FSCD.2022.1.

18 C. Kop. A weakly monotonic, logically constrained, HORPO-variant. Technical report, Radboud University, 2024. doi:10.48550/arXiv.2406.18493.

19 C. Kop and N. Nishida. Term rewriting with logical constraints. In P. Fontaine, C. Ringeissen, and R. A. Schmidt, editors, *Proc. FroCoS*, pages 343–358, 2013. doi:10.1007/978-3-642-40885-4_24.

20 C. Kop and D. Vale. The Cora analyzer. URL: https://github.com/hezzel/cora.

21 C. Kop and D. Vale. hezzel/cora: artifact for MFCS 2024. doi:10.5281/zenodo.12551027.

22 C. Kop and D. Vale. Tuple interpretations for higher-order complexity. In N. Kobayashi, editor, *Proc. FSCD*, pages 31:1–31:22, 2021. doi:10.4230/LIPIcs.FSCD.2021.31.

23 C. Kop and F. van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *LMCS*, 8(2):10:1–10:51, 2012. doi:10.2168/lmcs-8(2:10)2012.

24 K. Kusakari and M. Sakai. Enhancing dependency pair method using strong computability in simply-typed term rewriting. *AAECC*, 18(5):407–431, 2007. doi:10.1007/s00200-007-0046-9.

25 A. Matsumi, N. Nishida, M. Kojima, and D. Shin. On singleton self-loop removal for termination of LCTRSs with bit-vector arithmetic. In A. Yamada, editor, *Proc. WST*, 2023. doi:10.48550/arXiv.2307.14094.

26 P. W. O'Hearn. Continuous reasoning: scaling the impact of formal methods. In A. Dawar and E. Grädel, editors, *Proc. LICS*, pages 13–25, 2018. doi:10.1145/3209108.3209109.

27 M. R. K. K. Rao. Completeness of hierarchical combinations of term rewriting systems. In R. K. Shyamasundar, editor, *Proc. FSTTCS*, pages 125–138, 1993. doi:10.1007/3-540-57529-4_48.

28 M. R. K. K. Rao. Simple termination of hierarchical combinations of term rewriting systems. In M. Hagiya and J. C. Mitchell, editors, *Proc. TACS*, pages 203–223, 1994. doi:10.1007/3-540-57887-0_97.

29 M. R. K. K. Rao. Semi-completeness of hierarchical and super-hierarchical combinations of term rewriting systems. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *Proc. CAAP*, pages 379–393, 1995. doi:10.1007/3-540-59293-8_208.

30 M. Sakai and K. Kusakari. On dependency pair method for proving termination of higher-order rewrite systems. *IEICE Trans. Inf. Syst.*, E88-D(3):583–593, 2005. doi:10.1093/ietisy/e88-d.3.583.

31 M. Sakai, Y. Watanabe, and T. Sakabe. An extension of the dependency pair method for proving termination of higher-order rewrite systems. *IEICE Trans. Inf. Syst.*, E84-D(8):1025–1032, 2001. URL: https://search.ieice.org/bin/summary.php?id=e84-d_8_1025.

32 R. Tarjan. Depth-first search and linear graph algorithms. *SICOMP*, 1(2):146–160, 1972. doi:10.1137/0201010.

33 R. Thiemann. *The DP Framework for Proving Termination of Term Rewriting.* PhD thesis, RWTH Aachen University, 2007. URL: http://cl-informatik.uibk.ac.at/users/thiemann/paper/diss.pdf.

34 J. C. van de Pol. *Termination of Higher-Order Rewrite Systems.* PhD thesis, Utrecht University, 1996. URL: https://www.cs.au.dk/~jaco/papers/thesis.pdf.

35 A. Yamada. Tuple interpretations for termination of term rewriting. *JAR*, 66:667–688, 2022. doi:10.1007/s10817-022-09640-4.

36 H. Zantema. Termination of term rewriting: interpretation and type elimination. *JSC*, 17(1):23–50, 1994. doi:10.1006/jsco.1994.1003.