Higher-Order Constrained Dependency Pairs for (Universal) Computability

Liye Guo ORCID
Radboud University, Nijmegen, The Netherlands

Kasper Hagens ORCID
Radboud University, Nijmegen, The Netherlands

Cynthia Kop ORCID
Radboud University, Nijmegen, The Netherlands

Deivid Vale ORCID
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 hierarchical or 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.CVIT.2016.23

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

1 Introduction

Logically constrained simply-typed term rewriting systems (LCSTRSs) [11] are a formalism of higher-order term rewriting with logical constraints (built on its first-order counterpart [18]). Proposed for program analysis, LCSTRSs offer a flexible representation of programs since—unlike traditional rewriting—they can natively represent primitive data types such as (arbitrary-precision or fixed-width) integers and floating-point numbers. Without compromising ability to directly reason with 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 [11] 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., [29, 20, 28, 2]). Modern termination analyzers rely heavily on dependency pairs.

In higher-order termination analysis, dependency pairs take two forms: the dynamic [29, 20] and the static [28, 2, 21, 6]. This paper concentrates on the *static* variation, building on the definitions in [6, 21]. Dependency pairs for first-order rewriting with logical constraints have been informally defined by the third author [15], which we also build upon.

For program analysis, the traditional notion of termination can be inefficient, and arguably insufficient. It assumes the full program is known, and analyzed at once: a *closed-world* analysis. This means that even small programs that happen to use large standard libraries require a sophisticated analysis; and local changes in a large, previously verified program, require the entire analysis to be redone. As O'Hearn argues in [23] (though in a different

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

context), studying *open-world* analysis instead opens up many applications. In particular, it seems practically highly desirable to analyze termination of standard libraries, or modules in a larger program, 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, a program $\mathsf{app}\ (\mathsf{lam}\ f) \to f$ 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 program encodes the untyped lambda-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 to analyze these properties.

Various modular aspects of term rewriting have been studied by the community. Our scenario roughly corresponds to hierarchical combinations [25, 26, 27, 5], where parts of programs are analyzed separately, potentially using different analysis techniques. 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 present a first definition of *dependency pairs* for higher-order constrained TRSs (Section 3). Since the prior work for first-order rewriting we build on [15] was never formally published, this is also a first DP approach for logically constrained TRSs.
- We define a *dependency pair framework* for termination analysis, and provide five *dependency pair processors* to simplify termination problems in this framework (Section 4).
- We extend the notion of *hierarchical combinations* [25, 26, 27, 5] to LCSTRSs and define both *universal* and *public computability*. We fine-tune the DP framework to support these properties, and provide two new processors for public computability (Section 5). This allows the DP framework to be used for open-world and compositional analysis.
- We have implemented the DP framework for both termination and public computability in our open-source tool 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 [11]. 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 [6]. 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 for instance $t_0\ t_1\ t_2$ stands for $((t_0\ t_1)\ t_2)$.

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

The set $\mathrm{Var}(t)$ of variables in a term t 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 $([\![\cdot]\!]_A)_{A \in \mathcal{T}_\vartheta}$ where $[\![\cdot]\!]_A$ 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 $([\![\cdot]\!]_A)_{A \in \mathcal{T}_\vartheta}$, we extend each indexed mapping $[\![\cdot]\!]_B$ to one that assigns to each *ground theory term* of type B an element of $\mathfrak{X}_B$ by letting $[\![t_0\ t_1]\!]_B$ be $[\![t_0]\!]_{A \to B}([\![t_1]\!]_A)$. We write just $[\![\cdot]\!]$ 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 $\mathbb{Z}$, $[\![\cdot]\!]_{\mathsf{int}}$ be the identity mapping and $[\![-]\!]$ 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. Formally, if $\square$ is a special terminal symbol and A a type, a context is an element $C[]$ of $T(\mathcal{F}, \mathcal{V} \cup \{\square : A\})$ in which $\square$ occurs exactly once. Given a term $s : A$, we denote $C[s]$ for the term obtained by replacing $\square$ in $C[]$ by s.

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 $s_0 \neq t$; that is, $s = C[t]$ for $C[]$ which does not take the form $C'[\square\ t_1]$. We write $s \triangleright t$ if $s \trianglerighteq t$ and $s \neq t$.

Constrained Rewriting. Constrained rewriting requires the theory sort bool: we henceforth assume that $\mathsf{bool} \in \mathcal{S}_\vartheta$, $\{\, \mathfrak{f}, \mathfrak{t} \,\} \subseteq \mathcal{F}_\vartheta$, $\mathfrak{X}_{\mathsf{bool}} = \{\, 0, 1 \,\}$, $[\![\mathfrak{f}]\!]_{\mathsf{bool}} = 0$ 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 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* $\rightarrow_\mathcal{R} \subseteq T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V})$ such that $t \rightarrow_\mathcal{R} t'$ if and only if there exist a context $C[]$ and terms s, s' such that $t = C[s]$, $t' = C[s']$ and one of the following conditions is true:

1. $s = \ell\sigma$ and $s' = r\sigma$ for some $\ell \rightarrow r\ [\varphi] \in \mathcal{R}$ and substitution σ which respects $\ell \rightarrow r\ [\varphi]$.
2. $s = f\ v_1 \cdots v_n$ and $s' = v'$ for values $v_1 : A_1, \ldots, v_n : A_n, v' : B$ and theory symbol $f : A_1 \rightarrow \cdots \rightarrow A_n \rightarrow B$ with $n > 0$ such that $[\![f\ v_1 \cdots v_n]\!] = [\![v']\!]$.

If $t \rightarrow_\mathcal{R} t'$ due to the second condition, we also write $t \rightarrow_\kappa t'$ and call it a *calculation* step. When no ambiguity arises, we may simply write $\rightarrow$ for $\rightarrow_\mathcal{R}$. Let $s{\downarrow_\kappa}$ denote the result of maximally reducing a term s using calculation steps (e.g., $f\ (1 + (2 + 3)){\downarrow_\kappa} = f\ 6$).

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

▶ **Example 2.** The following LCSTRS implements the factorial function using continuations:

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

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

2.2 Accessibility and Computability

Accessibility. We assume given a *sort ordering* $\succsim$: a quasi-ordering 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 $\mathcal{T}$: for a sort A and a type $B = B_1 \rightarrow \cdots \rightarrow B_n \rightarrow C$ with C a sort and $n \geq 0$, we let $A \succsim_+ B$ if $A \succsim C$ and $A \succ_- B_i$ for all i; and we let $A \succ_- B$ if $A \succ C$ and $A \succsim_+ B_i$ for all i.

Given a function symbol $f : A_1 \rightarrow \cdots \rightarrow A_n \rightarrow B$ where B is a sort, the set of *accessible argument positions* of f is defined as $\mathrm{Acc}(f) = \{\, 1 \leq i \leq n \mid B \succsim_+ A_i \,\}$. 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 $f\ s_1 \cdots s_m \rightarrow r\ [\varphi] \in \mathcal{R}$ 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 (by equating all sorts in $\succsim$) if for all $f\ s_1 \cdots s_m \rightarrow r\ [\varphi] \in \mathcal{R}$, for all $i \in \{1, \ldots, m\}$: all strict subterms of s_i (i.e., t with $s_i \triangleright t$) have base type. Rewrite rules for common higher-order functions, e.g., map and fold, usually fit this criterion.

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

The system $\{\, \mathsf{app}\ (\mathsf{lam}\ f) \rightarrow f \,\}$ 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 $\rightarrow_\mathcal{R}$ 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 $\rightarrow_\mathcal{R}$) term of type A.
2. Given terms s and t such that $s \rightarrow_\mathcal{R} t$, 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 $s \rightarrow_\mathcal{R} t$, so is s.

Given an RC-set I for $\to_{\mathcal{R}}$, a term t_0 is *I-computable* if either the type of t_0 is a sort A and $t_0 \in I_A$, 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 [6]). *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 s and t both have 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 $\forall A \in \mathcal{S}$: $t \in \mathbb{C}_A$ iff $t : A$ is terminating with respect to $\to_{\mathcal{R}} \cup \Rrightarrow_{\mathbb{C}}$, and for all t' with $t \to_{\mathcal{R}}^* t'$, if $t' = f\ t_1 \cdots t_n$ for some function symbol f, then t_i is $\mathbb{C}$-computable for all $i \in \mathrm{Acc}(f)$.

Using this definition, a term $f\ t_1 \cdots t_n$ is computable iff all its $\to_{\mathcal{R}}$-reducts and accessible arguments $\{t_i \mid i \in \mathrm{Acc}(f)\}$ are. We will consider $\mathbb{C}$-computability throughout this paper.

3 Static Dependency Pairs for LCSTRSs

Originally proposed for first-order unconstrained 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 are multiple higher-order generalizations, among which we follow the *static* branch [21, 6]. As we will see in Section 5, this approach adapts 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 short. For each defined symbol $f : A_1 \to \cdots \to A_n \to B$ with B a sort, 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 is a set of variables such that $\mathrm{Var}(\varphi) \subseteq L$. For a rule $\ell \to r\ [\varphi]$, let $\mathrm{SDP}(\ell \to r\ [\varphi])$ denote the set of SDPs taking the 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 will see its usefulness in Section 4.4, as it gives us more freedom to manipulate DPs. 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 $\mathsf{gcd} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$.

$$\mathsf{gcdlist} \to \mathsf{fold}\ \mathsf{gcd}\ 0 \qquad \mathsf{fold}\ f\ y\ \mathsf{nil} \to y \qquad \mathsf{fold}\ f\ y\ (\mathsf{cons}\ x\ l) \to f\ x\ (\mathsf{fold}\ f\ y\ l)$$

$$\begin{array}{llll} \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ (-m)\ n & [m < 0] & \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ m\ (-n) & [n < 0] \\ \mathsf{gcd}\ m\ 0 \to m & [m \geq 0] & \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ n\ (m \bmod n) & [m \geq 0 \land n > 0] \end{array}$$

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 [6, 21].

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 (see Appendix A):

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

4 The Constrained DP Framework

In this section, we present several techniques based on SDPs, each as a *DP processor*; formally, we call this collection of DP processors the *constrained (static) DP framework*. In general, a DP framework [9, 6] 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 reachability between dependency pairs. While this graph is not computable in general, we follow the usual convention and use an (over-)approximation:

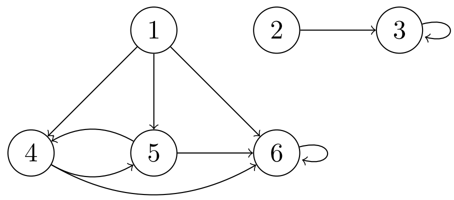
▶ **Definition 12.** *Given a set $\mathcal{P}$ of SDPs, a* graph approximation *G_θ for $\mathcal{P}$ is a finite directed graph such that θ maps the elements of $\mathcal{P}$ to the vertices of G_θ, and there is an edge from p_0 to p_1 if $(p_0, \sigma_0), (p_1, \sigma_1)$ is a $(\mathcal{P}, \mathcal{R})$-chain for some substitutions σ_0 and σ_1.*

Note that it is allowed for the graph approximation to have additional edges. Note also that, while we have allowed a DP problem $\mathcal{P}$ to be infinite in principle, in practice we typically only deal with a finite set of SDPs. Then, we can safely let θ be a bijection.

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 an approximation (G_θ, θ) of the DP graph of $\mathcal{P}$ 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 SDP($\mathcal{R}$) is given to the right. If a graph processor produces this graph as the approximation, it will return the set of DP problems $\{ \{ 3 \}, \{ 4, 5 \}, \{ 6 \} \}$.



Implementation. To compute a graph approximation, we adapt the common CAP approach [8, 32] by taking theories into account. The use of theories allows us to for instance *not* have an edge from (6) to (4) in the graph for Example 7.

We assume given a finite set of dependency pairs, and let $\theta(p) = p$ (i.e., the nodes of the approximation are just the DPs of $\mathcal{P}$). To test if there is an edge from $t \Rightarrow s\ [\varphi \mid L]$ to $t' \Rightarrow s'\ [\varphi' \mid L']$, where the latter SDP is renamed to have distinct variables from the former, we use an SMT solver to compute satisfiability of $\varphi \wedge \varphi' \wedge \zeta(s, t)$, where $\zeta(u, v)$ is given by:

- $\mathfrak{f}$ if $u = f\ u_1 \cdots u_n$ and $v = g\ v_1 \cdots v_m$ with $f \neq g$, if $f \in \mathcal{F}_\vartheta$ then v is not a value, and: there is no rule in $\mathcal{R}$ of the form $f\ \ell_1 \cdots \ell_k \to r\ [\psi]$ with $n \geq k$, and
- $\zeta(u_1, v_1) \wedge \cdots \wedge \zeta(u_n, v_n)$ if $u = f\ u_1 \cdots u_n$, $v = f\ v_1 \cdots v_n$ and there is no rule in $\mathcal{R}$ of the form $f\ \ell_1 \cdots \ell_k \to r\ [\psi]$ with $n \geq k$
- $u = v$ if $u \in T(\mathcal{F}_\vartheta, L)$ and $v \in T(\mathcal{F}_\vartheta, \mathcal{V})$ (and we are not in the cases above)
- $\mathfrak{t}$ in all other cases.

Note that "there is no rule in $\mathcal{R}$ of the form ..." can happen if f is a constructor (or symbol $f^\sharp$), theory symbol, or partially applied defined symbol. For example, since $m \geq 0 \wedge n > 0 \wedge m' < 0 \wedge n = m' \wedge (m \bmod n) = n'$ is unsatisfiable, there is no edge from (6) to (4).

Strongly connected components may be computed using Tarjan's SCC algorithm [31]).

4.2 The Subterm Criterion

The subterm criterion [13, 21] handles structural recursion and allows us to remove decreasing SDPs without considering rewrite rules in $\mathcal{R}$. We start with defining projections:

▶ **Definition 15.** *Let* 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* 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 16.** *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 17.** Following Example 14, a subterm criterion processor is applicable to $\{ 3 \}$. Let $\nu(\mathsf{fold}^\sharp) = 3$; then $\bar{\nu}(\mathsf{fold}^\sharp\ f\ y\ (\mathsf{cons}\ x\ l)) = \mathsf{cons}\ x\ l \rhd l = \bar{\nu}(\mathsf{fold}^\sharp\ f\ y\ l)$. The processor returns $\{ \emptyset \}$, and the empty DP problem can (trivially) be removed by a graph processor.

Implementation. The search for a suitable projection function can be done through SMT, and is standard: we use integer variables $N_{f^\sharp}$ for all $f^\sharp \in \mathrm{heads}(\mathcal{P})$ to represent $\nu(f^\sharp)$, and a Boolean variable strict_p for each $p \in \mathcal{P}$, and encode the requirements per DP.

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. For this processor, we assume that $\mathsf{int} \in \mathcal{S}_\vartheta$ and that $\mathcal{F}_\vartheta$ contains symbols $\geq, >: \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$ and $\wedge : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$ that are interpreted in the natural way.

▶ **Definition 18.** *Given a set $\mathcal{P}$ of SDPs, for all $f^\sharp \in \mathrm{heads}(\mathcal{P})$ (see Definition 15) 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 19.** *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 for all substitutions σ that map variables in $\mathrm{Var}(\varphi) \cup \mathrm{Var}(\varphi')$ to values: if $[\![\varphi\sigma]\!] = 1$ then $[\![\varphi'\sigma]\!] = 1$. 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 20.** 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 $m \geq 0 \wedge n > 0 \models n \geq 0 \wedge n > m \bmod n$. Then the integer mapping processor returns $\{\emptyset\}$, and the empty DP problem can (trivially) be removed by a graph processor.

Implementation. There are several ways to implement this processor. In our tool, we generate a number of candidate interpretations from the constraints, and use an encoding to SMT to select one candidate for each $f^\sharp \in \mathrm{heads}(\mathcal{P})$ that satisfies the requirements of Definition 19. Candidates are for instance all functions $\mathcal{J}(f^\sharp) = x_{f^\sharp,i}$, and candidates obtained from the constraint (e.g., for an SDP $f^\sharp\ x\ y \Rightarrow g^\sharp\ x\ (y+1)\ [y < x]$, we generate $\mathcal{J}(f^\sharp) = x_{f^\sharp,1} - x_{f^\sharp,2} - 1$ because $y < x$ implies $x - y - 1 \geq 0$).

4.4 Theory Arguments

The integer mapping processor has a clear limitation: what if some variables do not occur in the set L? This arises in the last remaining DP problem from Example 7: $\{4, 5\}$. This problem is clearly finite, but we cannot apply the integer mapping processor since $\iota(\mathsf{gcd}^\sharp) = \emptyset$.

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

In our running example, the problem 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, then we *can* derive that m must be instantiated by a ground theory term (we call such an argument a *theory argument*). We explore a way of propagating this information.

▶ **Definition 21.** *A* theory argument (position) mapping τ *for a set* $\mathcal{P}$ *of SDPs is a mapping from* heads($\mathcal{P}$) *(see Definition 15) to subsets of* $\mathbb{Z}$ *such that* $\tau(f^\sharp) \subseteq \{\, 1 \le i \le m \mid A_i \in \mathcal{S}_\vartheta \,\}$ *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 22.** *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 23.** Following Example 14, a theory argument processor is applicable to $\{\, 4, 5 \,\}$. Let $\tau(\mathsf{gcd}^\sharp) = \{\, 1 \,\}$, so τ fixes $\{\, 4 \,\}$. Then the processor returns $\{\{4, (7)\ \mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ m\ (-n)\ [n < 0 \mid \{\, m, n \,\}]\},\ \{5\}\}$. The integer mapping processor with $\mathcal{J}(\mathsf{gcd}^\sharp) = -x_{\mathsf{gcd}^\sharp,1}$ removes (4) from $\{\, 4, 7 \,\}$; then $\{\, 7 \,\}$ and $\{\, 5 \,\}$ are easily removed by a graph processor.

Implementation. To find a valid theory argument mapping, we can simply start by setting $\tau(f^\sharp) = \{1, \ldots, m\}$ for all $f^\sharp$, choose one DP to fix, and then iteratively remove arguments that violate the restrictions, until nothing further needs to be done.

4.5 Reduction Pairs

Although we did not need it for our running example, we also present a variant of the *Reduction Pair processor*, which is at the heart of most unconstrained termination provers.

▶ **Definition 24.** *A* constrained relation *is a set* R *of tuples* (s, t, φ, L)*, denoted* $s\ R_\varphi^L\ t$*, where* s *and* t *are terms of the same type,* φ *is a constraint, and* L *is a set of variables. We say a binary relation* R' *on terms* covers R *if* $s\ R_\varphi^L\ t$ *implies that* $(s\sigma)\downarrow_\kappa\ R'\ (t\varphi)\downarrow_\kappa$ *for any substitution* σ *that respects* φ *and maps all* $x \in L$ *to ground theory terms.*

A constrained reduction pair *is a pair* $(\succeq, \succ)$ *of constrained relations such that there exist a* reflexive *relation* $\sqsupseteq$ *that covers* $\succeq$ *and a* well-founded *relation* $\sqsupset$ *that covers* $\succ$ *such that* $\to_\kappa \subseteq \sqsupseteq$ *and* $\sqsupseteq \cdot \sqsupset \subseteq \sqsupset^+$ *and* $s \sqsupseteq t$ *implies* $C[s] \sqsupseteq C[t]$ *(for every appropriately-typed context).*

▶ **Definition 25.** *A* reduction pair 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 a constrained reduction pair* $(\succeq, \succ)$ *exists with (a)* $s^\sharp \succ_\varphi^L t^\sharp$ *for* $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L] \in \mathcal{P}'$*, (b)* $s^\sharp \succeq_\varphi^L t^\sharp$ *for* $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L] \in \mathcal{P}$*, and (c)* $\ell \succeq_\varphi^{(\mathrm{Var}(r)\setminus\mathrm{Var}(\ell))\cup\mathrm{Var}(\varphi)} r$ *for* $\ell \to r\ [\varphi] \in \mathcal{R}$*.*

While in unconstrained rewriting a variety of reduction pairs exist, this is not yet the case in constrained higher-order rewriting: the only definition so far is a limited version of the higher-order recursive path ordering [11]. To illustrate the practicality of the definition, we adapted the Horpo variant of [11] to a weakly monotonic reduction pair [12] (and implemented it in our tool using the technique described in [11]), but this is still a prototype definition.

We have included this processor because its existence allows us to start designing reduction pairs for use in the DP framework. In particular, as unconstrained reduction pairs can be used as the covering pair $(\sqsupseteq, \sqsupset)$, it is likely that many unconstrained reduction pairs (such as stronger Horpo variants and weakly monotonic algebras) can be adapted.

▶ **Theorem 26** (see Appendix B)**.** *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 to analyze termination 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 [25, 26, 27, 5] in terms of LCSTRSs:

▶ **Definition 27.** *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 28.** *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 29.** *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 the sorts of $\mathcal{R}_0 \cup \mathcal{R}_1$ (i.e., $\succsim'$ coincides with $\succsim$ on the sorts occurring 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 based on the same notion.

▶ **Example 30.** 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 31.** *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 32.** *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 sort ordering $\succsim'$ which extends $\succsim$ over sorts in $\mathcal{R}_0 \cup \mathcal{R}_1$.*

While this result itself is not surprising and its proof (see Appendix C) is standard, it is not so obvious how it can be used. The key observation which enables us to use the DP framework for public computability is that of 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 an approximation, a graph processor does not have to know the 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 the system $\mathcal{R}$ in Section 4 is the hierarchical combination $\mathcal{R}_0 \cup \mathcal{R}_1$.

First, we generalize the definition of a DP problem:

▶ **Definition 33.** *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 of the original graph, subterm criterion, integer mapping and theory argument processors *proc*, the updated processor that maps $(\mathcal{P}, \mathtt{p})$ to $\{(\mathcal{P}', \mathfrak{an}) \mid \mathcal{P}' \in \mathit{proc}(\mathcal{P})\}$ is sound for universal DP problems. For theory argument processors, also the processor that maps $(\mathcal{P}, \mathfrak{pu})$ to $\{\, (\{\, \bar{\tau}(p) \,|\, p \in \mathcal{P} \,\}, \mathfrak{pu}) \,\}$ if the theory argument mapping τ fixes all public SDPs in $\mathcal{P}$ is sound. 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 34.** *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 35.** *A* reachability processor *assigns to a DP problem $(\mathcal{P}, \mathfrak{pu})$ the singleton $\{\, (\{\, p \in \mathcal{P} \,|\, \theta(p)$ is reachable from $\theta(p_0)$ for some public SDP $p_0 \,\}, \mathfrak{pu}) \,\}$ for some approximation (G_θ, θ) of the DP graph of $\mathcal{P}$.*

These two processors are naturally used in combination: the constraint modification processor can split an SDP into multiple smaller ones, some of which can then be removed by a reachability argument. In our tool, we particularly use this to replace a DP with constraint $u \neq v$ by two SDPs with constraints $u > v$ and $u < v$ (see Example 36), and similar for $u \vee v$.

▶ **Example 36.** Consider an alternative implementation of the factorial function of Example 2, which has SDPs (1) $\mathsf{fact}^\sharp\ n\ k \Rightarrow \mathsf{fact}^\sharp\ (n-1)\ (\mathsf{comp}\ k\ ((*)\ n))\ [n \neq 0]$ and (2) $\mathsf{fact}^\sharp\ n\ k \Rightarrow \mathsf{comp}\ k\ ((*)\ n)\ [n \neq 0]$ and (3) $\mathsf{init}^\sharp\ k \Rightarrow \mathsf{fact}^\sharp\ 42\ k$, with fact a hidden symbol. Note that, without regarding the hidden symbols, this DP problem is not finite, as there is an infinite chain starting in $(1, [n := -1, k := \mathsf{id}])$. A constraint modification processor can be used to replace (1) by two new SDPs: (1a) $\mathsf{fact}^\sharp\ n\ k \Rightarrow \mathsf{fact}^\sharp\ (n-1)\ (\mathsf{comp}\ k\ ((*)\ n))\ [n < 0]$ and (1b) $\mathsf{fact}^\sharp\ n\ k \Rightarrow \mathsf{fact}^\sharp\ (n-1)\ (\mathsf{comp}\ k\ ((*)\ n))\ [n > 0]$. Using a reachability processor, we can remove (1a). This leaves only (1b), (2) and (3), which are easily handled using the graph processor and integer mapping processor.

▶ **Theorem 37** (see Appendix C)**.** *All the DP processors defined in Section 5 are sound.*

6 Experiments and Future Work

All results in this paper have been implemented in our open-source tool Cora, available at `https://github.com/hezzel/cora/` We have evaluated Cora on three groups of experiments, the results of which are given to the right.

	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 test considers the examples from this paper and several other LC(S)TRS benchmarks we have collected. The second test considers all λ-free problems from the higher-order category of the TPDB [4]. The third test considers problems from the first-order “integer TRS innermost” category. The computability test analyses public computability; since there are no hidden symbols in the TPDB, the main difference with the termination check is that the reduction pair processor is disabled. A full evaluation page is available at:

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

Unsurprisingly, Cora is substantially weaker than Wanda [16] on unconstrained higher-order benchmarks, or AProVE [7] 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 tools that have had years of development. However, for having only a handful of simple techniques, we believe that these results show a solid foundation.

Future Work. Moreover, much of the existing techniques used in the analysis of integer TRSs and higher-order TRS are likely to be adaptable to our setting, leaving many encouraging avenues for further development. We highlight the most important ones.

- Usable rules with respect to an argument filtering [8, 17]: to effectively use reduction pairs, being able to discard some rules is essential (especially for universal computability, if we can discard the unknown rules). Closely related, there is a clear benefit to extending more reduction pairs such as weakly monotonic algebras [34, 24], tuple interpretations [19, 33] and more sophisticated path orderings [3], all of which have higher-order defiitions.
- Transformation techniques, such as narrowing, or chaining DPs together (as used for instance for integer transition systems, [7, Sec. 3.1]). This could also be a step towards using the constrained DP framework for non-termination.
- Handling innermost or call-by-value reduction strategies. Several functional programming languages use call-by-value evaluation, and using this restriction may allow for a more powerful analysis. In the first-order DP framework there is ample work on the innermost strategy to build on (see, e.g., [9, 8]).
- Theory-specific processors for popular theories other than integers; e.g., bitvectors [22].

7 Conclusion

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 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.

4 Community. Termination Problem DataBase. 2023. URL: https://github.com/TermCOMP/TPDB.

5 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.

6 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.

7 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. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

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

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.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.

11 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.

12 L. Guo and C. Kop. A weakly monotonic, logically constrained, horpo-variant. Technical report, Radboud University, 2024. URL: https://github.com/hezzel/cora/tree/master/documentation/cora/techniques.

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 *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. Felty, editor, *Proc. FSCD*, pages 1:1–1:17, 2022. doi:10.4230/LIPIcs.FSCD.2022.1.

18 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.

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

20 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.

21 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.

22 A. Matsumi, N. Nishida, 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.

23 P. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *Proc. LICS*, pages 13–25, 2018. doi:10.1145/3209108.3209109.

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

25 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.

26 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.

27 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.

28 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.

29 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.

30 W. W. Tait. Intensional interpretations of functionals of finite type I. *JSL*, 32(2):198–212, 1967. doi:10.2307/2271658.

31 R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), 1972. doi:10.1137/0201010.

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

33 A. Yamada. Tuple interpretations for termination of term rewriting. *J. Automated Reasoning*, 66(4):667–688, 2022. doi:10.1007/s10817-022-09640-4.

34 H. Zantema. Termination of term rewriting: interpretation and type elimination. *J. Symbolic Computation*, 17:23–50, 1994. doi:10.1006/jsco.1994.1003.

A A Proof Sketch for Theorem 10

While Theorem 10 can be proved directly in the standard way, we instead point out its close connection to Theorem 32, which is a more general version of Theorem 10 and will be proved in full detail (see Appendix C). Below we show how to adapt the proof of Theorem 32.

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

Proof Sketch. Assume that $\mathcal{R}$ is non-terminating. By definition, there exists a non-terminating term u. Since all $\mathbb{C}$-computable terms are terminating, u is not $\mathbb{C}$-computable. We refer to the proof of Theorem 32, take $\mathcal{R}_0 = \mathcal{R}$ and $\mathcal{R}_1 = \emptyset$, and assume an empty set of hidden symbols in $\mathcal{R}_0$. Following the construction in the proof, we thus get an infinite computable $(\mathrm{SDP}(\mathcal{R}), \mathcal{R})$-chain. So the non-existence of such a chain implies the termination of $\mathcal{R}$. ◁

B Proofs for Section 4

We split up the proof of Theorem 26 into proofs for the individual processors.

▶ **Theorem 38.** *Graph processors are sound.*

Proof. By definition of DP graph, any $(\mathcal{P}, \mathcal{R})$-chain induces a path in the DP graph G of $\mathcal{P}$. By definition of graph homomorphism, this is directly mapped to a path in the approximation G_θ. Since the approximation only has finitely many vertices, the path of an infinite $(\mathcal{P}, \mathcal{R})$-chain must touch at least some vertices infinitely often.

Let $v_1, \ldots, v_n$ be these dependency pairs. Since all other vertices are only touched finitely often, eventually the path *only* touches these vertices; that is, the chain has an infinite tail containing only DPs p with $\theta(p) \in \{v_1, \ldots, v_n\}$.

Since each v_i occurs infinitely often, there is a path from each v_i to each v_j in G_θ. Thus, all v_i must be in the same strongly connected component. ◀

We did not formally state the following result in the text, but it is implied:

▶ **Lemma 39.** *The* Cap*-based approach described in the implementation part of Section 4.1 indeed yields an approximation of DP graph of $\mathcal{P}$. This holds whether we consider the graph for $\mathcal{R}$, or for any extension $\mathcal{R} \cup \mathcal{R}_1$.*

Proof. Let $t \Rightarrow s\ [\varphi \mid L]$ and $t \Rightarrow s\ [\varphi \mid L]$ be DPs with no shared variables. There should be an edge from $t \Rightarrow s\ [\varphi \mid L]$ to $t \Rightarrow s\ [\varphi \mid L]$ in a graph approximation if there exists a substitution σ such that (a) σ maps all variables in L and L' to ground theory terms, (b) $[\![\varphi\sigma]\!] = [\![\varphi'\sigma]\!] = 1$, and (c) $s\sigma \to_{\mathcal{R}}^* t'\sigma$. As we only need an approximation, false positives are allowed. Thus we must see, if such σ exists, then $\varphi \wedge \varphi' \wedge \zeta(s, t)$ is satisfiable. We claim that it is satisfiable by the valuation that maps x to $[\![\sigma(x)]\!]$ whenever $\sigma(x)$ is a ground theory term. With this valuation, certainly φ and φ' are satisfied. To complete the proof, we show by induction on u that if $u\sigma \to_{\mathcal{R} \cup \mathcal{R}_1}^* v\sigma$ then this valuation satisfies $\zeta(u, v)$.

- If $u = f\ u_1 \cdots u_n$ and there is a rule in $\mathcal{R}$ that could potentially reduce $u\sigma$ at the top, then the left-hand side of such a rule must apply f to at most n arguments. In this case, $\zeta(u, v)$ yields $\mathfrak{t}$, so there is nothing to prove (recall that false positives are allowed).
- If $u = f\ u_1 \cdots u_n$ with f not a theory symbol, and there is no rule in $\mathcal{R}$ that can reduce $u\sigma$ at the top, then since f occurs in the original signature, there is no rule in $\mathcal{R}_1$ that can reduce u at the top either (by definition of an extension). Hence, $v\sigma$ must have a shape $f\ t_1 \cdots t_n$. So, the case $v = g\ u_1 \cdots u_m$ with $f \neq g$ cannot occur, and if $v = f\ v_1 \cdots v_n$ then by the induction hypothesis the valuation must satisfy $\zeta(u_i, v_i)$ for all i. In all other cases, $\zeta(u, v)$ yields $\mathfrak{t}$, so again, there is nothing to prove.
- If $u = f\ u_1 \cdots u_n$ with f a theory symbol, and there is a rule in $\mathcal{R}$ that could potentially reduce $u\sigma$ at the top, then $\zeta(u, v)$ yields $\mathfrak{t}$ as in the first case. If there is not, then the only way to reduce $u\sigma$ at the top is using a calculation rule. Hence, $v\sigma$ must have a shape $f\ t_1 \cdots t_n$, or be a value.
  - If v is not a value, but does have a form $g\ u_1 \cdots u_m$ (or $f\ v_1 \cdots v_n$), then $v\sigma$ is not a value, so we complete as we did in the second case.
  - If v is a value or variable, and all variables in u occur in L, then note that $u\sigma$ is a ground theory term, and as left-hand sides of rules cannot be theory terms, the only way to reduce a ground theory term uses the calculation rules, which preserves the value of the term. Hence, $u = v$ must hold in the given valuation.
  - In all other cases, $\zeta(u, v)$ yields $\mathfrak{t}$, so there is nothing to prove.
- If u is a variable in L, then $u\sigma$ is a ground theory term. As above, we conclude that $u = v$ must hold in the given valuation.
- If u is a variable not in L then $\zeta(u, v)$ yields $\mathfrak{t}$, so there is nothing to prove. The same holds if $u = x\ u_1 \cdots u_n$ with $n > 0$, since in this case x cannot occur in L by definition of a constraint. ◀

▶ Remark 40. We do not have any clever logic for the case where $v = x\ v_1 \cdots v_m$, because this case does not occur in practice (as left-hand sides of DPs are in practice always patterns). It would not be hard to add additional cases for this if new DP processors were to be defined that broke the pattern property, however.

▶ **Theorem 41.** *Subterm criterion processors are sound.*

Proof. We first observe: if $u \trianglerighteq \cdot \to_{\mathcal{R}}^* v$ then $u \to_{\mathcal{R}}^* C[v]$ for some context C. Hence, if $t_i \trianglerighteq \cdot \to_{\mathcal{R}}^* t_{i+1}$ for all i but t_1 is terminating, then eventually this sequence stops using $\to_{\mathcal{R}}$ steps: there exists N such that for all $i > N$: $t_i \trianglerighteq t_{i+1}$. But *strict* subterm steps are also terminating, since they decrease the size of the term. So, eventually all $\trianglerighteq$ steps are really equalities. Thus, there exists M such that for all $i > M$: $t_i = t_{i+1}$.

Now, any infinite computable $\mathcal{P}$-chain yields an infinite sequence of steps $\bar{\nu}(s_i^\sharp \sigma_i) \trianglerighteq \bar{\nu}(t_i^\sharp \sigma_i) \to^* \bar{\nu}(s_{i+1}^\sharp \sigma_{i+1})$. Since the immediate arguments of all $t_i^\sharp \sigma_i$ are computable, and therefore terminating, such a sequence can only have finitely many steps where the $\trianglerighteq$ step is strict (as we observed above). Thus, there is an infinite tail of the chain using only SDPs that are not in $\mathcal{P}'$. ◀

▶ **Theorem 42.** *Integer mapping processors are sound.*

Proof. We first observe: if $i \in \iota(g^\sharp)$ and σ respects an SDP $f^\sharp\ s_1 \cdots s_m \Rightarrow g^\sharp\ t_1 \cdots t_n\ [\varphi \mid L]$, then by definition of $\iota()$ and "respect", $t_i\sigma$ is a ground theory term, and since the left-hand sides of rules in $\mathcal{R}$ cannot be theory terms, any $\to_{\mathcal{R}}^*$ reduct of $t_i\sigma$ can only be obtained with $\to_\kappa$, which does not change the value.

Hence, in an infinite chain $[s_j^\sharp \Rightarrow t_j^\sharp\ [\varphi_j \mid L_j] \mid j \in \mathbb{N}]$, for all j we have that $\bar{\mathcal{J}}(s_j^\sharp)\sigma_j$ and $\bar{\mathcal{J}}(t_j^\sharp)\sigma_j$ are necessarily ground theory terms, and $[\![\bar{\mathcal{J}}(t_j^\sharp)\sigma_j]\!] = [\![\bar{\mathcal{J}}(t_{j+1}^\sharp)\sigma_{j+1}]\!]$

Let $\sigma_j^\downarrow$ be the substitution that maps each x in the domain of σ_j to $\sigma_j(x)\downarrow_\kappa$. Then σ_j maps the variables in each constraint and in $\bar{\mathcal{J}}(s_j^\sharp)$ and $\bar{\mathcal{J}}(t_j^\sharp)$ to values (all these variables occur in L_j, so have a theory sort).

Since σ_j respects φ_j, we have that $[\![\varphi_j\sigma_j]\!] = [\![\varphi_j\sigma_j^\downarrow]\!] = 1$, so by assumption $[\![\bar{\mathcal{J}}(s_j^\sharp)\sigma_j]\!] = [\![\bar{\mathcal{J}}(s_j^\sharp)\sigma_j^\downarrow]\!] \geq [\![\bar{\mathcal{J}}(t_j^\sharp)\sigma_j^\downarrow]\!] = [\![\bar{\mathcal{J}}(t_j^\sharp)\sigma_j]\!]$ for all $s_j^\sharp \Rightarrow t_j^\sharp\ [\varphi_j \mid L_j] \in \mathcal{P} \setminus \mathcal{P}'$, and for $s_j^\sharp \Rightarrow t_j^\sharp\ [\varphi_j \mid L_j] \in \mathcal{P}'$ we even have both $[\![\bar{\mathcal{J}}(s_j^\sharp)\sigma_j]\!] > [\![\bar{\mathcal{J}}(t_j^\sharp)\sigma_j]\!]$ and $[\![\bar{\mathcal{J}}(s_j^\sharp)\sigma_j]\!] > 0$.

Since only finitely many decreasing steps can be done before reaching 0, any infinite $\mathcal{P}$-chain must have an infinite tail not using the elements of $\mathcal{P}'$. ◀

▶ **Theorem 43.** *Theory argument processors are sound.*

Proof. Let us say that a term $f^\sharp\ s_1 \cdots s_m$ respects τ if s_i is a ground theory term for every $i \in \tau(f^\sharp)$. Note that:

1. If t respects τ and $t \to_{\mathcal{R}} s$ then also s respects τ, because ground theory terms can only be rewritten using $\to_\kappa$ (since left-hand sides of rules may not be theory terms).
2. For any dependency pair $f^\sharp\ s_1 \cdots s_n \Rightarrow g^\sharp\ t_1 \cdots t_m\ [\varphi \mid L]$ and substitution σ that respects this DP: if $f^\sharp\ s_1 \cdots s_n\sigma$ respects τ then $\sigma(x)$ is a ground theory term for any $x \in L \cup \{y \in \mathrm{Var}(s_i) \mid i \in \tau(f)$. Therefore, by definition of a theory arguments mapping, also $t_j\sigma$ is a ground theory term for any $j \in \tau(g)$, so $g^\sharp\ t_1 \cdots t_m$ respects τ as well.
3. If τ fixes a dependency pair $s \Rightarrow t\ [\varphi \mid L]$, then for any substitution σ that respects this DP, $t\sigma$ respects τ.

Hence, in an infinite $(\mathcal{P}, \mathcal{R})$-chain $[(s_i \Rightarrow t_i\ [\varphi_i \mid L_i], \sigma_i) \mid i \in \mathbb{N}]$, if any DP $s_i \Rightarrow t_i\ [\varphi_i \mid L_i]$ fixes τ, then by (3) $t_i\sigma_i$ respects τ, so by (1) $s_{i+1}\sigma_{i+1}$ respects τ, and by (2) also $t_{i+1}\sigma_{i+1}$ respects τ. Hence, the chain has an infinite tail such that each σ_j ($j > i$) respects the SDP $\bar{\tau}(s_j \Rightarrow t_j\ [\varphi_j \mid L_j])$.

Thus, if τ fixes $\mathcal{P}'$ there are two possibilities to create an infinite $(\mathcal{P}, \mathcal{R})$-chain: either the chain does not use any DP in $\mathcal{P}'$—so it is a $(\mathcal{P} \setminus \mathcal{P}', \mathcal{R})$-chain—or it has an infinite tail that is a $(\{\bar{\tau}(p) \mid p \in \mathcal{P}\}, \mathcal{R})$-chain.

Moreover, if it is given that the first DP in the chain is an element of $\mathcal{P}'$ (which is the case in the public computability setting if $\mathcal{P}'$ includes all public dependency pairs) then the chain *is* a $(\{\bar{\tau}(p) \mid p \in \mathcal{P}\}, \mathcal{R})$-chain. ◀

▶ **Theorem 44.** *Reduction pair processors are sound.*

Proof. Suppose a reduction pair with the given properties is given, and let $(\sqsupseteq, \sqsupset)$ be the covering pair. We observe:

- If $s \to_{\mathcal{R}} t$ then $s\downarrow_\kappa \sqsupseteq^* t\downarrow_\kappa$.
  Proof: by induction on the form of s.
  - If s is a ground theory term, then note that no rule from $\mathcal{R}$ applies, since rules may not be theory terms. Hence, the reduction is with a $\to_\kappa$ step, and we have $s\downarrow_\kappa = t\downarrow_\kappa$.
  - Otherwise, if $s = \ell\sigma$ and $t = r\sigma$ for some rule $\ell \to r\ [\varphi]$ and substitution σ that respects this rule, then by (c) and the assumption that $\sqsupseteq$ covers $\succeq$ we have $s\downarrow_\kappa \sqsupseteq t\downarrow_\kappa$.
  - Otherwise, $s = s_1\ s_2$, and either $t = s_1\ t_2$ with $s_2 \to_{\mathcal{R}} t_2$ or $t = t_1\ s_2$ with $s_1 \to_{\mathcal{R}} t_1$. We consider only the second option; the first is similar. Since we excluded the case that s is a ground theory term, s does not $\to_\kappa$-reduce at the top, so $s\downarrow_\kappa = (s_1\downarrow_\kappa)\ (s_2\downarrow_\kappa)$. By the induction hypothesis, $s_1\downarrow_\kappa \sqsupseteq^* t_1\downarrow_\kappa$. Thus, by monotonicity of $\sqsupseteq$ we have $s\downarrow_\kappa \sqsupseteq (t_1\downarrow_\kappa)\ (s_2\downarrow_\kappa)$, which $\sqsupseteq^* ((t_1\downarrow_\kappa)\ s_2\downarrow_\kappa))\downarrow_\kappa = t\downarrow_\kappa$ because $\sqsupseteq$ includes $\to_\kappa$ (this covers the case where t is a ground theory term even though s is not).
- If $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L] \in \mathcal{P}'$, and σ is a substitution that respects $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L]$, then $(s\sigma)\downarrow_\kappa \sqsupset (t\sigma)\downarrow_\kappa$, and similarly, if $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L] \in \mathcal{P} \setminus \mathcal{P}'$, and σ is a substitution that respects $s^\sharp \Rightarrow t^\sharp\ [\varphi \mid L]$, then $(s\sigma)\downarrow_\kappa \sqsupseteq (t\sigma)\downarrow_\kappa$.
  Proof: immediately by (a), (b) and the definition of "covers".

Hence, any infinite $(\mathcal{P}, \mathcal{R})$-chain induces an infinite sequence of $\sqsupseteq^*$ and $\sqsupset$ steps, and if any step in $\mathcal{P}'$ occurs infinitely often, then $\sqsupset$ occurs infinitely often. Since $\sqsupseteq \cdot \sqsupset$ is included in $\sqsupset^+$, this yields an infinitely decreasing $\sqsupset^+$ sequence, which contradicts well-foundedness of $\sqsupset$. Hence, we see that the steps in $\mathcal{P}'$ can occur at most finitely often, and there is an infinite tail of the dependency chain using only pairs in $\mathcal{P} \setminus \mathcal{P}'$. ◀

C Proofs for Section 5

For the properties of $\mathbb{C}$-computability, see Appendix A[1] of [6]. In order to prove Theorem 32, we first present two lemmas.

▶ **Lemma 45.** *Undefined function symbols are $\mathbb{C}$-computable.*

Proof. Given an LCSTRS $\mathcal{R}$ and an undefined function symbol $f : A_1 \to \cdots \to A_n \to B$ where B is a sort, if f was uncomputable, there would be computable terms $t_1, \ldots, t_n$ making $f\ t_1 \cdots t_n$ uncomputable. Since f is undefined, any reduct of $f\ t_1 \cdots t_n$ must be either $f\ t'_1 \cdots t'_n$ where $t_i \to_{\mathcal{R}} t'_i$ for all i or a value (when f is a theory symbol). By definition, $f\ t_1 \cdots t_n \in \mathbb{C}$, which contradicts its uncomputability. ◀

[1] https://doi.org/10.48550/arXiv.1902.06733

▶ **Lemma 46.** *Given an AFP system $\mathcal{R}_0$ with sort ordering $\succsim$, an extension $\mathcal{R}_1$ of $\mathcal{R}_0$ and a sort ordering $\succsim'$ which extends $\succsim$ over sorts in $\mathcal{R}_0 \cup \mathcal{R}_1$, for each defined symbol $f : A_1 \to \cdots \to A_m \to B$ in $\mathcal{R}_0$ where B is a sort, if $f\ s_1 \cdots s_m$ is not $\mathbb{C}$-computable in $\mathcal{R}_0 \cup \mathcal{R}_1$ with $\succsim'$ but s_i is for all i, there exist an SDP $f^\sharp\ s'_1 \cdots s'_m \Rightarrow g^\sharp\ t_1 \cdots t_n\ [\varphi \mid L] \in \mathrm{SDP}(\mathcal{R}_0)$ and a substitution σ such that (1) $s_i \to^*_{\mathcal{R}_0 \cup \mathcal{R}_1} s'_i\sigma$ for all i, (2) σ respects $f^\sharp\ s'_1 \cdots s'_m \Rightarrow g^\sharp\ t_1 \cdots t_n\ [\varphi \mid L]$, and (3) $g\ (t_1\sigma) \cdots (t_n\sigma)$ is not $\mathbb{C}$-computable in $\mathcal{R}_0 \cup \mathcal{R}_1$ with $\succsim'$ but $u\sigma_i$ is for all i and u such that $t_i \trianglerighteq u$.*

Proof. We consider $\mathbb{C}$-computability in $\mathcal{R}_0 \cup \mathcal{R}_1$ with $\succsim'$. If all the reducts of $f\ s_1 \cdots s_m$ were either $f\ s'_1 \cdots s'_m$ where $s_i \to^*_{\mathcal{R}_0 \cup \mathcal{R}_1} s'_i$ for all i or a value (when f is a theory symbol), $f\ s_1 \cdots s_m$ would be computable. Hence, there exist a rewrite rule $f\ s'_1 \cdots s'_p \to r\ [\varphi] \in \mathcal{R}_0$ (f cannot be defined in $\mathcal{R}_1$) and a substitution σ' such that $s_i \to^*_{\mathcal{R}_0 \cup \mathcal{R}_1} s'_i\sigma'$ for all $i \le p$ and σ' respects the rewrite rule; $(r\sigma')\ s_{p+1} \cdots s_m$ is thus a reduct of $f\ s_1 \cdots s_m$. There is at least one such reduct that is uncomputable—otherwise, $f\ s_1 \cdots s_m$ would be computable. Let $(r\sigma')\ s_{p+1} \cdots s_m$ be uncomputable, and therefore so is $r\sigma'$.

Take a minimal subterm t of r such that $t\sigma'$ is uncomputable. If $t = x\ t_1 \cdots t_q$ for some variable x, $\sigma'(x)$ is either a value or an accessible subterm of $s'_k\sigma'$ for some k because $\mathcal{R}_0$ is AFP. Either way, $\sigma'(x)$ is computable. Due to the minimality of t, $t_i\sigma'$ is computable for all i, which implies that $t\sigma' = \sigma'(x)\ (t_1\sigma') \cdots (t_q\sigma')$ is computable. This contradiction shows that $t = g\ t_1 \cdots t_q$ for some function symbol g in $\mathcal{R}_0$. And g must be a defined symbol.

Now we have an SDP $f^\sharp\ s'_1 \cdots s'_p\ x_{p+1} \cdots x_m \Rightarrow g^\sharp\ t_1 \cdots t_q\ y_{q+1} \cdots y_n\ [\varphi \mid \mathrm{Var}(\varphi) \cup (\mathrm{Var}(r) \setminus \mathrm{Var}(f\ s'_1 \cdots s'_p))] \in \mathrm{SDP}(\mathcal{R}_0)$. Because $t\sigma'$ is uncomputable, there exist computable terms $t'_{q+1}, \ldots, t'_n$ such that $(t\sigma')\ t'_{q+1} \cdots t'_n = g\ (t_1\sigma') \cdots (t_q\sigma')\ t'_{q+1} \cdots t'_n$ is uncomputable. Let σ be the substitution such that $\sigma(x_i) = s_i$ for all $i > p$, $\sigma(y_i) = t'_i$ for all $i > q$, and $\sigma(z) = \sigma'(z)$ for any other variable z. Let s'_i denote x_i for all $i > p$, let t_i denote y_i for all $i > q$, and let L denote $\mathrm{Var}(\varphi) \cup (\mathrm{Var}(r) \setminus \mathrm{Var}(f\ s'_1 \cdots s'_p))$, then $f^\sharp\ s'_1 \cdots s'_m \Rightarrow g^\sharp\ t_1 \cdots t_n\ [\varphi \mid L]$ and σ satisfy all the requirements. ◀

▶ **Theorem 32.** *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 sort ordering $\succsim'$ which extends $\succsim$ over sorts in $\mathcal{R}_0 \cup \mathcal{R}_1$.*

Proof. Assume that $\mathcal{R}_0$ is not publicly computable. By definition, there exist a public extension $\mathcal{R}_1$ of $\mathcal{R}_0$, a sort ordering $\succsim'$ which extends $\succsim$ over sorts in $\mathcal{R}_0 \cup \mathcal{R}_1$ and a public term u of $\mathcal{R}_0$ such that u is not $\mathbb{C}$-computable in $\mathcal{R}_0 \cup \mathcal{R}_1$ with $\succsim'$. We consider $\mathbb{C}$-computability in $\mathcal{R}_0 \cup \mathcal{R}_1$ with $\succsim'$. Take a minimal uncomputable subterm s of u, then s must take the form $f\ s_1 \cdots s_k$ where f is a defined symbol in $\mathcal{R}_0$ and s_i is computable for all i. Let the type of f be denoted by $A_1 \to \cdots \to A_m \to B$ where B is a sort. Because $s = f\ s_1 \cdots s_k$ is uncomputable, there exist computable terms $s_{k+1}, \ldots, s_m$ of $\mathcal{R}_0 \cup \mathcal{R}_1$ such that $s\ s_{k+1} \cdots s_m = f\ s_1 \cdots s_m$ is uncomputable.

Repeatedly applying Lemma 46—first on $f\ s_1 \cdots s_m$, then on $g\ (t_1\sigma) \cdots (t_n\sigma)$ and so on—we thus get an infinite computable $(\mathrm{SDP}(\mathcal{R}_0), \mathcal{R}_0 \cup \mathcal{R}_1)$-chain. By construction, f is not a hidden symbol, and therefore this chain is public. So the non-existence of such a chain implies the public computability of $\mathcal{R}_0$. ◀

We split up the proof of Theorem 37 into proofs for the individual processors.

▶ **Theorem 47.** *Constraint modification processors are sound.*

Proof. If $\mathcal{P}'$ covers $\mathcal{P}$, every computable $(\mathcal{P}, \mathcal{R}_0 \cup \mathcal{R}_1)$-chain corresponds to a computable $(\mathcal{P}', \mathcal{R}_0 \cup \mathcal{R}_1)$-chain which has the same length and the same heading symbol. ◀

▶ **Theorem 48.** *Reachability processors are sound.*

Proof. Approximations over-approximate the DP graph and SDPs that are unreachable from any public SDP cannot contribute to public chains. ◀

▶ **Theorem 49.** *The modified theory argument processors are sound.*

Proof. We refer to the proof of Theorem 43, and in particular the final observation: "if it is given that the first DP in the chain is an element of $\mathcal{P}'$ (which is the case in the public computability setting if $\mathcal{P}'$ includes all public dependency pairs) then the chain *is* a $(\{\bar{\tau}(p) \mid p \in \mathcal{P}\}, \mathcal{R})$-chain." As the root symbol of the first DP in the chain is unchanged, the chain is still public. ◀