

BIROn - Birkbeck Institutional Research Online


Fuhs, Carsten and Guo, L. and Kop, C. (2025) An Innermost DP Framework for Constrained Higher-Order Rewriting. In: Fernández, M. (ed.) Proceedings of the 10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025). Leibniz International Proceedings in Informatics. Dagstuhl Publishing, 20:1-20:24.

Downloaded from: <https://eprints.bbk.ac.uk/id/eprint/55675/>

Usage Guidelines:

Please refer to usage guidelines at <https://eprints.bbk.ac.uk/policies.html> or alternatively contact lib-eprints@bbk.ac.uk.

An Innermost DP Framework for Constrained Higher-Order Rewriting

Carsten Fuhs  

Birkbeck, University of London, The United Kingdom

Liye Guo  

Radboud University, Nijmegen, The Netherlands

Cynthia Kop  

Radboud University, Nijmegen, The Netherlands

Abstract

Logically constrained simply-typed term rewriting systems (LCSTRSs) are a higher-order formalism for program analysis with support for primitive data types. The termination problem of LCSTRSs has been studied so far in the setting of full rewriting. This paper modifies the higher-order constrained dependency pair framework to prove innermost termination, which corresponds to the termination of programs under call by value. We also show that the notion of universal computability with respect to innermost rewriting can be effectively handled in the modified, innermost framework, which lays the foundation for open-world termination analysis of programs under call by value via LCSTRSs.

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

Keywords and phrases Higher-order term rewriting, constrained rewriting, innermost termination, call by value, open-world analysis, dependency pairs

Digital Object Identifier 10.4230/LIPIcs.FSCD.2025.17

Supplementary Material *Software (Source Code)*: <https://github.com/hezzel/cora> [27]

Software (Source Code): <https://zenodo.org/records/15318964> [28]

Funding *Liye Guo*: NWO VI.Vidi.193.075, project “CHORPE”

Cynthia Kop: NWO VI.Vidi.193.075, project “CHORPE”

Acknowledgements We thank the reviewers for helpful comments that allowed us to improve the presentation.

1 Introduction

In the study of term rewriting, *termination* has been an active area of research for decades. Hundreds of different termination techniques have been developed, along with a variety of (fully automatic) termination analyzers that compete against each other in an annual competition [6]. Many of those techniques have been adapted to different styles of term rewriting (e.g., context-sensitive, relative, constrained and higher-order).

In a recent work [22], we have introduced *logically constrained simply-typed term rewriting systems* (LCSTRSs): a variant of term rewriting that incorporates both higher-order terms and primitive data types such as integers and bit vectors. This formalism is proposed to be a stepping stone toward functional programming languages: by adapting analysis techniques from traditional term rewriting to LCSTRSs, we obtain many of the ingredients needed by the analysis of functional programs, without limiting ourselves to a particular language.

Our long-term goal is to use LCSTRSs as an intermediate verification language in a two-step process to prove correctness properties (e.g., termination, reachability and equivalence) of a program P written in a real-world programming language with higher-order features



© Carsten Fuhs, Liye Guo, and Cynthia Kop;

licensed under Creative Commons License CC-BY 4.0

10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025).

Editor: Maribel Fernández; Article No. 17; pp. 17:1–17:23

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

(e.g., OCaml and Scala): (1) soundly translate P to an LCSTRS \mathcal{R}_P (i.e., if \mathcal{R}_P is, say, terminating, then so is P), and (2) analyze \mathcal{R}_P . This approach has been successfully applied across paradigms to multiple languages, such as Prolog [44, 17], Haskell [16], Java [39] and C [14], via various flavors of term rewriting. We consider LCSTRSs a highly suitable intermediate language that allows for a direct representation of many programming language features. We particularly study termination, both for its own relevance, and due to the fact that it allows the reduction relation to be used as a decreasing measure in induction, which is a powerful aid to proving many other properties.

Most modern termination techniques for term rewriting are defined within the *dependency pair (DP) framework* [3, 18], which essentially studies groups of recursive function calls. In [21], a higher-order variant of this framework [35, 36, 34, 13] has been adapted to LCSTRSs, and its usefulness in open-world termination analysis has also been established through the notion of *universal computability*. However, this method still faces limitations: many commonly used DP processors (termination techniques within the DP framework) have not yet been extended to the higher-order and constrained setting, and whereas the first-order DP framework can deal with both full rewriting and innermost rewriting, most higher-order versions—including the one for LCSTRSs—only concern full rewriting. The incapability to handle innermost rewriting is particularly unfortunate since many functional languages use call-by-value evaluation, which largely corresponds to innermost rewriting.

In this paper, we define the first innermost DP framework for LCSTRSs. We shall present DP processors that explicitly benefit from innermost or call-by-value rewriting.

Contributions. We start in Section 2 with the preliminaries on LCSTRSs and the notion of computability, which is fundamental to static dependency pairs. Then this paper’s contributions follow:

- In Section 3, we discuss the innermost and the call-by-value strategies, and present a transformation that allows us to better utilize the call-by-value setting.
- In Section 4, we recall the notion of a static dependency pair for LCSTRSs [21], and extend it by introducing *call-by-value* dependency pairs and *innermost* chains.
- In Section 5, we present an *innermost DP framework*, which can be used to prove innermost (and call-by-value) termination through the use of various DP processors. We first review three classes of existing DP processors defined for full rewriting, and see how they adapt to the innermost setting. Then we define three more, specifically for the innermost DP framework:
 - In Section 5.2, following the idea of *chaining* from first-order rewriting with integer constraints [10, 11, 15], we propose a class of DP processors that merge call-by-value dependency pairs. These DP processors can be particularly useful in the analysis of automatically generated systems, which often give rise to a large body of dependency pairs representing intermediate steps of computation.
 - In Section 5.3, we extend the idea of *usable rules* [3, 19, 24]. While this idea has been applied to higher-order rewriting [2, 40], logical constraints still pose challenges. In the innermost setting, we are able to define this class of DP processors in their most powerful form, which permanently removes rewrite rules from a DP problem.
 - In Section 5.4, we show how usable rules with respect to an *argument filtering* may be defined, and how this technique makes *first-order* reduction pairs applicable. This way a large number of first-order techniques can be employed for higher-order termination without a higher-order modification; to be applied to LCSTRSs, those reduction pairs only need to adapt to logical constraints.

- In Section 6, we discuss the notion of *universal computability* [21] with respect to innermost rewriting. We will see that the employment of usable rules significantly increases the potential of the innermost DP framework to analyze universal computability: now we can use fully-fledged reduction pair processors for open-world termination analysis, and thereby harness one of the main benefits of the DP framework in this practical setting.
- We have implemented all the results in our open-source analyzer *Cora*. The implementation and the evaluation thereof are described in Section 7.

2 Preliminaries

This section collects the preliminaries from the literature. First, we recall the definition of an LCSTRS [22].¹ Then, we recollect from [13] the definition of computability (with accessibility), particularly under the strategies. We borrow several definitions from the phrasing in [21].

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 $\mathcal{T} ::= \mathcal{S} \mid (\mathcal{T} \rightarrow \mathcal{T})$. Right-associativity is assigned to \rightarrow 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 \mathcal{T} of *pre-terms* over \mathcal{F} and \mathcal{V} is generated by the grammar $\mathcal{T} ::= \mathcal{F} \mid \mathcal{V} \mid (\mathcal{T} \mathcal{T})$. 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 every function symbol and variable is assigned a unique type. Typing works as expected: if pre-terms t_0 and t_1 have types $A \rightarrow B$ 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 $\text{Var}(t)$ of variables in $t \in T(\mathcal{F}, \mathcal{V})$ is defined by $\text{Var}(f) = \emptyset$ for $f \in \mathcal{F}$, $\text{Var}(x) = \{x\}$ for $x \in \mathcal{V}$ and $\text{Var}(t_0 t_1) = \text{Var}(t_0) \cup \text{Var}(t_1)$. A term t is called *ground* if $\text{Var}(t) = \emptyset$.

For constrained rewriting, we make further assumptions. First, we assume there is a distinguished subset \mathcal{S}_θ of \mathcal{S} , called the set of *theory sorts*. The grammar $\mathcal{T}_\theta ::= \mathcal{S}_\theta \mid (\mathcal{S}_\theta \rightarrow \mathcal{T}_\theta)$ generates the set \mathcal{T}_θ of *theory types* over \mathcal{S}_θ . Note that a theory type is essentially a non-empty list of theory sorts. Next, we assume there is a distinguished subset \mathcal{F}_θ of \mathcal{F} , called the set of *theory symbols*, and the type of every theory symbol is in \mathcal{T}_θ , which means the type of any argument passed to a theory symbol is a theory sort. Theory symbols whose type is a theory sort are called *theory values*.² Elements of $T(\mathcal{F}_\theta, \mathcal{V})$ are called *theory terms*.

Theory symbols are interpreted in an underlying theory: given an \mathcal{S}_θ -indexed family of sets $(\mathfrak{X}_A)_{A \in \mathcal{S}_\theta}$, we extend it to a \mathcal{T}_θ -indexed family by letting $\mathfrak{X}_{A \rightarrow B}$ be the set of mappings from \mathfrak{X}_A to \mathfrak{X}_B ; an *interpretation* of theory symbols is a \mathcal{T}_θ -indexed family of mappings $(\llbracket \cdot \rrbracket_A)_{A \in \mathcal{T}_\theta}$ where $\llbracket \cdot \rrbracket_A$ assigns to each theory symbol of type A an element of \mathfrak{X}_A and is bijective if $A \in \mathcal{S}_\theta$. Given an interpretation of theory symbols $(\llbracket \cdot \rrbracket_A)_{A \in \mathcal{T}_\theta}$, 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 $\llbracket t_0 \rrbracket_{A \rightarrow B}(\llbracket t_1 \rrbracket_A)$. We write just $\llbracket \cdot \rrbracket$ when the type can be deduced.

¹ In contrast to [22] and following [21], we assume that ℓ is a pattern in every rewrite rule $\ell \rightarrow r$ [φ]. For notational convenience, we also assume that $\text{Var}(r) \setminus \text{Var}(\ell) \subseteq \text{Var}(\varphi)$, which can be guaranteed by a simple transformation and is also adopted in the second half of [22].

² Such theory symbols are simply called values in [22]. In this paper, we call them differently so that they are not to be confused with term values as in *call by value*.

► **Example 1.** Let \mathcal{S}_θ be $\{\text{int}\}$. Then $\text{int} \rightarrow \text{int} \rightarrow \text{int}$ is a theory type over \mathcal{S}_θ while $(\text{int} \rightarrow \text{int}) \rightarrow \text{int}$ is not. Let \mathcal{F}_θ be $\{-\} \cup \mathbb{Z}$ where $- : \text{int} \rightarrow \text{int} \rightarrow \text{int}$ and $n : \text{int}$ for all $n \in \mathbb{Z}$. The theory values are the elements of \mathbb{Z} . Let $\mathfrak{X}_{\text{int}}$ be \mathbb{Z} , $\llbracket \cdot \rrbracket_{\text{int}}$ be the identity mapping and $\llbracket - \rrbracket$ be the mapping $\lambda m. \lambda n. m - n$. The interpretation of $(-) 1$ is the mapping $\lambda n. 1 - n$ (note that functions in our setting are curried).

Type-preserving mappings from \mathcal{V} to $T(\mathcal{F}, \mathcal{V})$ are called *substitutions*. The *domain* of a substitution σ is the set $\text{dom}(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$. Let $[x_1 :- t_1, \dots, x_n :- t_n]$ denote the substitution σ such that $\text{dom}(\sigma) \subseteq \{x_1, \dots, x_n\}$ and $\sigma(x_i) = t_i$ for all i . 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 $(t_0 t_1)\sigma = (t_0\sigma) (t_1\sigma)$.

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

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

Constrained Rewriting. Constrained rewriting requires the theory sort **bool**: we henceforth assume that $\text{bool} \in \mathcal{S}_\theta$, $\{\mathfrak{f}, \mathfrak{t}\} \subseteq \mathcal{F}_\theta$, $\mathfrak{X}_{\text{bool}} = \{0, 1\}$, $\llbracket \mathfrak{f} \rrbracket_{\text{bool}} = 0$ and $\llbracket \mathfrak{t} \rrbracket_{\text{bool}} = 1$. Moreover, we require that \mathcal{F}_θ includes symbols $\wedge : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$, and for each sort $A \in \mathcal{S}_\theta$ also a symbol $\equiv_A : A \rightarrow A \rightarrow \text{bool}$, interpreted respectively as conjunction and equality operators.

A *logical constraint* is a theory term φ such that φ has type **bool** and the type of each variable in $\text{Var}(\varphi)$ is a theory sort. A (*constrained*) *rewrite rule* is a triple $\ell \rightarrow r [\varphi]$ where ℓ and r are terms which have the same type, φ is a logical constraint, $\text{Var}(r) \setminus \text{Var}(\ell) \subseteq \text{Var}(\varphi)$, and ℓ is a pattern that takes the form $\mathfrak{f} t_1 \dots t_n$ for some $\mathfrak{f} \in \mathcal{F}$ and contains at least one function symbol in $\mathcal{F} \setminus \mathcal{F}_\theta$. Here a *pattern* is a term whose subterms are either $\mathfrak{f} t_1 \dots t_n$ for some $\mathfrak{f} \in \mathcal{F}$ or a variable.³ A substitution σ is said to *respect* a logical constraint φ if $\sigma(x)$ is a theory value for all $x \in \text{Var}(\varphi)$ and $\llbracket \varphi\sigma \rrbracket = 1$; σ respects the rule $\ell \rightarrow r [\varphi]$ if it respects φ .

A *logically constrained simply-typed term rewriting system* (LCSTRS) collects the above data— \mathcal{S} , \mathcal{S}_θ , \mathcal{F} , \mathcal{F}_θ , \mathcal{V} , (\mathfrak{X}_A) and $\llbracket \cdot \rrbracket$ —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* $\rightarrow_{\mathcal{R}}$ over terms: $t \rightarrow_{\mathcal{R}} t'$ if and only if there exists a context $C[\]$ such that one of the following holds:

- (1) $t = C[\ell\sigma]$ and $t' = C[r\sigma]$ for some $\ell \rightarrow r [\varphi] \in \mathcal{R}$ and substitution σ which respects φ , or
- (2) $t = C[\mathfrak{f} v_1 \dots v_n]$ and $t' = C[v']$ for some theory symbol \mathfrak{f} and some theory values v_1, \dots, v_n, v' with $n > 0$ and $\llbracket \mathfrak{f} v_1 \dots v_n \rrbracket = \llbracket v' \rrbracket$.

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

A term t is in *normal form* if there is no term t' such that $t \rightarrow_{\mathcal{R}} t'$. Given an LCSTRS \mathcal{R} , the set $\mathcal{NF}(\mathcal{R})$ contains all terms that are in normal form with respect to $\rightarrow_{\mathcal{R}}$.

Given an LCSTRS \mathcal{R} , \mathfrak{f} is called a *defined symbol* if there is at least one rewrite rule of the form $\mathfrak{f} t_1 \dots t_n \rightarrow r [\varphi]$. Let \mathcal{D} denote the set of defined symbols. Theory values and function symbols in $\mathcal{F} \setminus (\mathcal{F}_\theta \cup \mathcal{D})$ are called *constructors*.

³ As usual in term rewriting, we do *not* require that each variable occurs at most once in a pattern.

► **Example 2.** Consider the following LCSTRS \mathcal{R} , where $\text{gcdlist} : \text{intlist} \rightarrow \text{int}$, $\text{fold} : (\text{int} \rightarrow \text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{intlist} \rightarrow \text{int}$ and $\text{gcd} : \text{int} \rightarrow \text{int} \rightarrow \text{int}$:

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

We use infix notation for some binary operators, and omit logical constraints that are \mathbf{t} . Here is a rewrite sequence: $\text{gcdlist } (\text{cons } (1 + 1) \ \text{nil}) \rightarrow_{\mathcal{R}} \text{fold gcd } 0 \ (\text{cons } (1 + 1) \ \text{nil}) \rightarrow_{\mathcal{R}} \text{gcd } (1 + 1) \ (\text{fold gcd } 0 \ \text{nil}) \rightarrow_{\mathcal{R}} \text{gcd } (1 + 1) \ 0 \rightarrow_{\kappa} \text{gcd } 2 \ 0 \rightarrow_{\mathcal{R}} 2$.

Innermost Rewriting. The *innermost rewrite relation* $\xrightarrow{i}_{\mathcal{R}}$ requires all the proper subterms of a redex to be in normal form: $t \xrightarrow{i}_{\mathcal{R}} t'$ if and only if either (1) there exist a context $C[\]$, a substitution σ and a rewrite rule $\ell \rightarrow r \ [\varphi] \in \mathcal{R}$ such that $t = C[\ell\sigma]$, $s \not\rightarrow_{\mathcal{R}} s'$ for any proper subterm s of $\ell\sigma$ and any $s', t' = C[r\sigma]$ and σ respects φ , or (2) $t \rightarrow_{\kappa} t'$.

Similarly, for an LCSTRS \mathcal{Q} we define the relation $\xrightarrow{e}_{\mathcal{R}}$ as reduction using $\rightarrow_{\mathcal{R}}$ where all proper subterms of a redex are in $\rightarrow_{\mathcal{Q}}$ -normal form: $t \xrightarrow{e}_{\mathcal{R}} t'$ if and only if either (1) there exist $C[\]$, σ and $\ell \rightarrow r \ [\varphi] \in \mathcal{R}$ such that $t = C[\ell\sigma]$, $s \not\rightarrow_{\mathcal{Q}} s'$ for any proper subterm s of $\ell\sigma$ and any $s', t' = C[r\sigma]$ and σ respects φ , or (2) $t \rightarrow_{\kappa} t'$. Note that $\xrightarrow{i}_{\mathcal{R}}$ is the same as $\xrightarrow{e}_{\mathcal{R}}$.

Call-By-Value Rewriting. We may further restrict a redex by requiring each proper subterm to be a ground term value. Here a *term value* is either a variable or a term $f \ v_1 \cdots v_n$ where f is a function symbol, v_i is a term value for all i , and (1) if there is a rule $f \ t_1 \cdots t_k \rightarrow r \ [\varphi] \in \mathcal{R}$, then $k > n$; (2) if f is a calculation symbol, then it takes at least $n + 1$ arguments (i.e., $f : A_1 \rightarrow \cdots \rightarrow A_{n+1} \rightarrow B$). In this paper, we will typically refer to term values as just *values*. By definition, all theory values are values, and all values are in normal form.

The definition of the *call-by-value rewrite relation* $\xrightarrow{v}_{\mathcal{R}}$ follows the pattern of $\xrightarrow{i}_{\mathcal{R}}$: $t \xrightarrow{v}_{\mathcal{R}} t'$ if and only if either (1) there exist a context $C[\]$, a substitution σ and a rewrite rule $\ell \rightarrow r \ [\varphi] \in \mathcal{R}$ such that $t = C[\ell\sigma]$, s is a ground value for each proper subterm s of $\ell\sigma$, $t' = C[r\sigma]$ and σ respects φ , or (2) $t \rightarrow_{\kappa} t'$.

► **Example 3.** The rewrite sequence in Example 2 is not innermost (and therefore not call-by-value). An example of a call-by-value rewrite sequence is: $\text{gcdlist } (\text{cons } (1 + 1) \ \text{nil}) \xrightarrow{v}_{\mathcal{R}} \text{fold gcd } 0 \ (\text{cons } (1 + 1) \ \text{nil}) \rightarrow_{\kappa} \text{fold gcd } 0 \ (\text{cons } 2 \ \text{nil}) \xrightarrow{v}_{\mathcal{R}} \text{gcd } 2 \ (\text{fold gcd } 0 \ \text{nil}) \xrightarrow{v}_{\mathcal{R}} \text{gcd } 2 \ 0 \xrightarrow{v}_{\mathcal{R}} 2$. Note that the redex in the first step is gcdlist , which does not have $1 + 1$ as subterm.

► **Example 4.** Consider the LCSTRS $\mathcal{R} = \{\text{hd } (\text{cons } x \ l) \rightarrow x, \text{tl } (\text{cons } x \ l) \rightarrow l\}$. Then the rewrite step $\text{hd } (\text{cons } 42 \ (\text{tl } \text{nil})) \xrightarrow{i}_{\mathcal{R}} 42$ is an innermost step, but not a call-by-value step. The reason is that the subterm $\text{tl } \text{nil}$ of the redex is in normal form, but not a value: innermost rewriting also allows for rewriting above function calls that are not defined on the given arguments; in a language like OCaml, the computation would abort with an error.

2.2 Accessibility and Computability

Accessibility. Assume given a *sort ordering*—a quasi-ordering \succsim over \mathcal{S} whose strict part $\succ = \succsim \setminus \preceq$ is well-founded. We inductively define two relations \succsim_+ and \succ_- over \mathcal{S} and \mathcal{T} : for a sort A and a type $B = B_1 \rightarrow \cdots \rightarrow B_n \rightarrow C$ where C is a sort and $n \geq 0$, $A \succsim_+ B$ if $A \succsim C$ and $A \succ_- B_i$ for all i , and $A \succ_- B$ 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 $\text{Acc}(f)$ of the *accessible argument positions* of f is defined as $\{i \mid B \succsim_+ A_i\}$. A term t is called an *accessible*

subterm of a term s , written as $s \succeq_{\text{acc}} t$, if either $s = t$, or $s = f \ s_1 \cdots s_n$ for some $f \in \mathcal{F}$ and there exists $k \in \text{Acc}(f)$ such that $s_k \succeq_{\text{acc}} t$. An LCSTRS \mathcal{R} is called *accessible function passing* (AFP) if there exists a sort ordering such that for all $f \ s_1 \cdots s_n \rightarrow r \ [\varphi] \in \mathcal{R}$ where $f \in \mathcal{F}$ and $x \in \text{Var}(r) \setminus \text{Var}(\varphi)$, there exists k such that $s_k \succeq_{\text{acc}} x$.

► **Example 5.** An LCSTRS \mathcal{R} is AFP (with \succeq equating all the sorts) if for all $f \ s_1 \cdots s_n \rightarrow r \ [\varphi] \in \mathcal{R}$ where $f \in \mathcal{F}$ and $i \in \{1, \dots, n\}$, the type of each proper subterm of s_i is a sort. This is typical for many common examples of higher-order programs manipulating first-order data; e.g., integer recursion or list folding, mapping or filtering; hence, Example 2 is AFP.

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

Consider $\{\text{app} (\text{lam } f) \rightarrow f\}$ where $\text{app} : \text{o} \rightarrow \text{o} \rightarrow \text{o}$ and $\text{lam} : (\text{o} \rightarrow \text{o}) \rightarrow \text{o}$. This system encodes the untyped lambda-calculus, with app serving as the application symbol and lam as a wrapper for abstractions. It is not AFP since $\text{o} \succ \text{o}$ 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 a type-preserving relation \rightarrow over terms—which may stand for $\rightarrow_{\mathcal{R}}$, $\xrightarrow{i}_{\mathcal{R}}$, but also $\xrightarrow{v}_{\mathcal{R}}$ —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) term of type A .
- (2) Given terms s and t such that $s \rightarrow t$, 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 $s \rightarrow t$, so is s .

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 $A \rightarrow B$ 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} :

► **Theorem 6** (see [13]). *Given a sort ordering and an RC-set I for \rightarrow , let \Rightarrow_I be the relation over terms such that $s \Rightarrow_I t$ 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 $k \in \text{Acc}(f)$ 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 \rightarrow such that $t \in \mathbb{C}_A$ if and only if $t : A$ is terminating with respect to $\rightarrow \cup \Rightarrow_{\mathbb{C}}$, and for all t' such that $t \rightarrow^ t'$, if $t' = f \ t_1 \cdots t_n$ for some function symbol f , t_i is \mathbb{C} -computable for all $i \in \text{Acc}(f)$.*

\mathbb{C} -computability with respect to the innermost reduction relation $\xrightarrow{i}_{\mathcal{R}}$ is at the heart of the soundness proofs for the DP framework defined in this paper (Theorems 14 and 36).

3 The Transformation of Call-By-Value Systems

Let us reflect on the shape of call-by-value LCSTRSs. First, we observe that if a pattern t is not a value, neither are its instances, i.e., $t\sigma$ is not a value for any substitution σ . Hence, under call by value, a rewrite rule $\ell \rightarrow r \ [\varphi]$ where not all the proper subterms of ℓ are values is never applicable, and we can exclude such rewrite rules without loss of generality.

Second, programming languages hardly allow the addition of constructors to pre-defined (in particular, primitive) data types, such as integers. Given an LCSTRS, in practice, we would also expect, for example, that integer literals are the only constructors for int . Formally, we call a theory sort B *inextensible* if every function symbol $f : A_1 \rightarrow \cdots \rightarrow A_n \rightarrow B$ is a theory symbol or a defined symbol. We do *not* require that all theory sorts are inextensible; rather, we assume that for each rewrite rule $\ell \rightarrow r \ [\varphi]$, all the variables in $\text{Var}(\ell)$ whose type is an inextensible theory sort are also in $\text{Var}(\varphi)$. We can do so without loss of generality

because such a variable can only be instantiated to a theory value under call by value. We only consider inextensible theory sorts in examples below.

We write φ, x_1, \dots, x_n for $\varphi \wedge x_1 \equiv x_1 \wedge \dots \wedge x_n \equiv x_n$, and conclude the above discussion:

► **Lemma 7.** *We apply the below transformation to an LCSTRS \mathcal{R} and let \mathcal{R}' be the outcome.*

- (1) *Remove all $\ell \rightarrow r [\varphi]$ where ℓ has a proper subterm that is not a value.*
 - (2) *Replace all remaining $\ell \rightarrow r [\varphi]$ with $\ell \rightarrow r [\varphi, x_1, \dots, x_n]$ where x_1, \dots, x_n are the variables in $\text{Var}(\ell) \setminus \text{Var}(\varphi)$ whose type is an inextensible theory sort.*
- Then $t \xrightarrow{\mathcal{R}} t'$ if and only if $t \xrightarrow{\mathcal{R}'} t'$ for all t and t' .*

► **Example 8.** The LCSTRS \mathcal{R} from Example 2 is transformed as follows: (1) no rewrite rules are removed, and (2) four rewrite rules are replaced by the ones below.

$$\begin{array}{lll} \text{fold } f \ y \ \text{nil} \rightarrow y & [\mathbf{t}, y] & \text{gcd } m \ n \rightarrow \text{gcd } (-m) \ n \quad [m < 0, n] \\ \text{fold } f \ y \ (\text{cons } x \ l) \rightarrow f \ x \ (\text{fold } f \ y \ l) & [\mathbf{t}, x, y] & \text{gcd } m \ n \rightarrow \text{gcd } m \ (-n) \quad [n < 0, m] \end{array}$$

We will refer to the result of the transformation as \mathcal{R}_{gcd} .

We are particularly interested in call-by-value termination. However, *innermost* termination is more commonly considered in the term rewriting community, and has been studied extensively for first-order term rewriting. In practice, its difference from call-by-value termination is often irrelevant, and their respective techniques are generally similar. Since all values are in normal form, $\xrightarrow{\mathcal{R}}$ is terminating if $\xrightarrow{\mathcal{R}'}$ is. Hence, we shall formulate our results in terms of innermost rewriting, for the sake of more generality and less bookkeeping.

4 Static Dependency Pairs and Chains

The dependency pair method [3] analyzes the recursive structure of function calls. Its variants are at the heart of most modern automatic termination analyzers for various styles of term rewriting. *Static* dependency pairs [35, 13], a higher-order generalization of the method, are adapted to LCSTRSs in [21]. Now we extend this adaptation for the evaluation strategies.

First, we recall a notation:

► **Definition 9.** *Given an LCSTRS \mathcal{R} , let $\mathcal{F}^\#$ be $\mathcal{F} \uplus \{\mathbf{f}^\# \mid \mathbf{f} \in \mathcal{D}\}$ where \mathcal{D} is the set of defined symbols in \mathcal{R} and $\mathbf{f}^\#$ is a fresh function symbol for all \mathbf{f} . Let \mathbf{dp} be a fresh sort, and for each defined symbol $\mathbf{f} : A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$ where $B \in \mathcal{S}$, we assign $\mathbf{f}^\# : A_1 \rightarrow \dots \rightarrow A_n \rightarrow \mathbf{dp}$. Given a term $t = \mathbf{f} \ t_1 \dots t_n \in T(\mathcal{F}, \mathcal{V})$ where $\mathbf{f} \in \mathcal{D}$, let $t^\#$ denote $\mathbf{f}^\# \ t_1 \dots t_n \in T(\mathcal{F}^\#, \mathcal{V})$.*

The definition of a static dependency pair is adapted as follows:

► **Definition 10.** *A static dependency pair (SDP) is a triple $s^\# \Rightarrow t^\# [\varphi]$ where $s^\#$ and $t^\#$ are terms of type \mathbf{dp} , and φ is a logical constraint. This SDP is considered call-by-value if all the proper subterms of $s^\#$ are values. For a rewrite rule $\ell \rightarrow r [\varphi]$, let $\text{SDP}(\ell \rightarrow r [\varphi])$ denote the set of SDPs of form $\ell^\# \ x_1 \dots x_m \Rightarrow \mathbf{g}^\# \ t_1 \dots t_q \ y_{q+1} \dots y_n [\varphi]$ such that*

- (1) $\ell^\# : A_1 \rightarrow \dots \rightarrow A_m \rightarrow \mathbf{dp}$ with fresh variables x_1, \dots, x_m ,
- (2) $r \ x_1 \dots x_m \sqsupseteq \mathbf{g} \ t_1 \dots t_q$ for $\mathbf{g} \in \mathcal{D}$, and
- (3) $\mathbf{g}^\# : B_1 \rightarrow \dots \rightarrow B_n \rightarrow \mathbf{dp}$ with fresh variables y_{q+1}, \dots, y_n .

Let $\text{SDP}(\mathcal{R})$ be $\bigcup_{\ell \rightarrow r [\varphi] \in \mathcal{R}} \text{SDP}(\ell \rightarrow r [\varphi])$. For $s^\# \Rightarrow t^\# [\mathbf{t}]$, we just write $s^\# \Rightarrow t^\#$.

Compared to the quadruple definition in [21], this definition omits the L component, because here its bookkeeping role can be assumed by the logical constraint.

Intuitively, every dependency pair in $\text{SDP}(\mathcal{R})$ represents a function call in \mathcal{R} . If \mathcal{R} is non-terminating, there must be an infinite “chain” of function calls whose *arguments* are terminating (in other words, the chain is “minimal”: all the proper subterms are terminating). To distinguish the head symbols of those minimal potentially non-terminating terms, we write f^\sharp instead of f at head positions in SDPs. Then calls to a defined symbol f (the original) can be assumed to terminate, which is shown separately via f^\sharp . And the non-existence of an infinite chain starting from a call to any f^\sharp in $\text{SDP}(\mathcal{R})$ implies the termination of \mathcal{R} .

► **Example 11.** Following Example 8, $\text{SDP}(\mathcal{R}_{\text{gcd}})$ consists of the following SDPs:

- | | |
|--|---|
| (1) $\text{gcdlist}^\sharp l' \Rightarrow \text{gcd}^\sharp m' n'$ | (4) $\text{gcd}^\sharp m n \Rightarrow \text{gcd}^\sharp m (-n) [n < 0, m]$ |
| (2) $\text{gcdlist}^\sharp l' \Rightarrow \text{fold}^\sharp \text{gcd } 0 l'$ | (5) $\text{gcd}^\sharp m n \Rightarrow \text{gcd}^\sharp n (m \bmod n) [m \geq 0 \wedge n > 0]$ |
| (3) $\text{gcd}^\sharp m n \Rightarrow \text{gcd}^\sharp (-m) n [m < 0, n]$ | (6) $\text{fold}^\sharp f y (\text{cons } x l) \Rightarrow \text{fold}^\sharp f y l [t, x, y]$ |

In this paper, a set \mathcal{R} of rewrite rules plays two roles: (1) it specifies how to rewrite arguments in SDPs, and (2) it determines which rewrite steps are innermost and which terms are values. In Sections 5 and 6, a given set \mathcal{R} in its first role may be modified (with rewrite rules removed). In this process, if we do not keep the original set, there can be unintended consequences for its second role. For example, consider $\mathcal{R} = \{f x \rightarrow r_1, a \rightarrow r_2\}$. Note that the first rewrite rule is not applicable to $f a$ with respect to innermost rewriting. Now if we remove the second rewrite rule from \mathcal{R} and consider the new set, the remaining rewrite rule will be applicable, and the term a will be a value, which is generally an unwanted change.

Hence, we keep a copy of the original set \mathcal{R} of rewrite rules to faithfully determine which rewrite steps are innermost and which terms are values. Following [18, 41], we let \mathcal{Q} denote this copy. In contrast to the literature, we consider \mathcal{Q} fixed throughout the analysis of a given system.⁴ Now we formalize the idea of a chain of function calls in the innermost setting:

► **Definition 12.** Given a set \mathcal{P} of SDPs and a set \mathcal{R} of rewrite rules, an innermost $(\mathcal{P}, \mathcal{R})$ -chain is a (finite or infinite) sequence $(s_0^\sharp \Rightarrow t_0^\sharp [\varphi_0], \sigma_0), (s_1^\sharp \Rightarrow t_1^\sharp [\varphi_1], \sigma_1), \dots$ such that for all i , $s_i^\sharp \Rightarrow t_i^\sharp [\varphi_i] \in \mathcal{P}$, σ_i is a substitution which respects φ_i , $s_i^\sharp \sigma_i$ is in normal form with respect to $\rightarrow_{\mathcal{Q}}$, and $t_{i-1}^\sharp \sigma_{i-1} \xrightarrow{\mathcal{Q}}^*_{\mathcal{R}} s_i^\sharp \sigma_i$ if $i > 0$.

► **Example 13.** Following Example 11, $(1, [l' := \text{nil}, m' := 24, n' := 18]), (5, [m := 24, n := 18]), (5, [m := 18, n := 6])$ is an innermost $(\text{SDP}(\mathcal{R}_{\text{gcd}}), \mathcal{R}_{\text{gcd}})$ -chain.

In the above definition, the requirement that $s_i^\sharp \sigma_i$ is in normal form implies that $\sigma_i(x)$ is in normal form for all $x \in \text{Var}(s_i^\sharp)$, including fresh variables such as l' in SDP (1). Hence, the call-by-value (and therefore innermost) rewrite sequence in Example 3 does not directly translate to an innermost $(\text{SDP}(\mathcal{R}), \mathcal{R})$ -chain. Nevertheless, we have the following result:

► **Theorem 14.** An AFP system \mathcal{R} is innermost (and therefore call-by-value) terminating if there exists no infinite innermost $(\text{SDP}(\mathcal{R}), \mathcal{R})$ -chain.

Proof Idea. The full proof is in Appendix A.2, and is very similar to its counterpart for termination with respect to full rewriting [21]: in a non-terminating LCSTRS, we can always identify a non-terminating base-type term $t = f t_1 \dots t_n$ such that all t_i are \mathbb{C} -computable; and from an infinite reduction $t \xrightarrow{i}_{\mathcal{R}} t' \xrightarrow{i}_{\mathcal{R}} t'' \xrightarrow{i}_{\mathcal{R}} \dots$ we can find s, u such that $s = f (t_1 \downarrow_{\mathcal{R}}) \dots (t_n \downarrow_{\mathcal{R}})$, (s^\sharp, u^\sharp) is an instance of a dependency pair, and u is again

⁴ A fixed set \mathcal{Q} suffices in this paper because our DP processors may remove rewrite rules, but may not add new rewrite rules or change the signature (e.g., with semantic labeling [45]).

non-terminating with \mathbb{C} -computable arguments. The primary difference from [21] is the need to ensure all variables are instantiated to normal forms, which for the fresh variables discussed above means that we must choose the right normal form that still yields non-termination. ◀

5 The Innermost DP Framework

In this section, we modify the constrained DP framework [21] to prove innermost termination. The DP framework is a collection of DP processors, each of which represents a technique.

► **Definition 15.** A DP problem is a pair $(\mathcal{P}, \mathcal{R})$ where \mathcal{P} is a set of SDPs and \mathcal{R} is a set of rewrite rules. If no infinite innermost $(\mathcal{P}, \mathcal{R})$ -chain exists, $(\mathcal{P}, \mathcal{R})$ is called finite. 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 $(\mathcal{P}, \mathcal{R})$ is finite whenever all the elements of $\rho(\mathcal{P}, \mathcal{R})$ are.

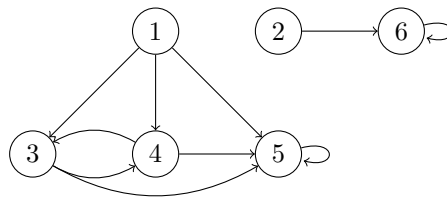
Theorem 14 tells us that an AFP system \mathcal{R} is innermost terminating if $(\text{SDP}(\mathcal{R}), \mathcal{R})$ is a finite DP problem. Given a collection of sound DP processors, we have the following procedure: (1) $\mathcal{Q} := \mathcal{R}$, $S := \{(\text{SDP}(\mathcal{R}), \mathcal{R})\}$; (2) while S contains a DP problem $(\mathcal{P}, \mathcal{R})$ to which some sound DP processor ρ is applicable, $S := (S \setminus \{(\mathcal{P}, \mathcal{R})\}) \cup \rho(\mathcal{P}, \mathcal{R})$. If this procedure ends with $S = \emptyset$, we can conclude that \mathcal{R} is innermost terminating.

5.1 Graph, Subterm Criterion and Integer Mapping Processors

Three classes of DP processors in [21] remain essentially unchanged in the innermost setting. Let us review their application to the system \mathcal{R}_{gcd} from Example 8, which is discussed by [21] in the form of Example 2. First, we consider a graph approximation for $(\text{SDP}(\mathcal{R}_{\text{gcd}}), \mathcal{R}_{\text{gcd}})$:

► **Definition 16.** For a DP problem $(\mathcal{P}, \mathcal{R})$, a graph approximation (G, θ) consists of a finite directed graph G and a mapping θ from \mathcal{P} to the vertices of G such that there is an edge from $\theta(p_0)$ to $\theta(p_1)$ whenever $(p_0, \sigma_0), (p_1, \sigma_1)$ is an innermost $(\mathcal{P}, \mathcal{R})$ -chain for some σ_0 and σ_1 .

Compared to [21], the main change is that we now check for *innermost* $(\mathcal{P}, \mathcal{R})$ -chains to determine which edges must exist. A graph approximation for $(\text{SDP}(\mathcal{R}_{\text{gcd}}), \mathcal{R}_{\text{gcd}})$ is in Figure 1. Despite the slightly different SDPs (due to the transformation of the call-by-value system), this graph approximation is the same as the one in [21].



■ **Figure 1** A graph approximation for $(\text{SDP}(\mathcal{R}_{\text{gcd}}), \mathcal{R}_{\text{gcd}})$.

Given a graph approximation, we can decompose the DP problem:

► **Definition 17.** Given a DP problem $(\mathcal{P}, \mathcal{R})$, a graph processor computes a graph approximation (G, θ) for $(\mathcal{P}, \mathcal{R})$ and the strongly connected components (SCCs) of G , then returns $\{(\{p \in \mathcal{P} \mid \theta(p) \text{ belongs to } S\}, \mathcal{R}) \mid S \text{ is a non-trivial SCC of } G\}$.

If a graph processor produces the graph approximation in Figure 1, it will return the set $\{(\{6\}, \mathcal{R}_{\text{gcd}}), (\{3, 4\}, \mathcal{R}_{\text{gcd}}), (\{5\}, \mathcal{R}_{\text{gcd}})\}$. Next, we observe that **fold** is defined by structural recursion, which can be handled by subterm criterion processors. Let $\text{heads}(\mathcal{P})$ denote the set of function symbols heading either side of an SDP in \mathcal{P} , and we have the following definition:

► **Definition 18.** A projection ν for a set \mathcal{P} of SDPs is a mapping from $\text{heads}(\mathcal{P})$ to integers such that $1 \leq \nu(f^\#) \leq n$ if $f^\# : A_1 \rightarrow \dots \rightarrow A_n \rightarrow \text{dp}$. Let $\bar{\nu}(f^\# t_1 \dots t_n)$ denote $t_{\nu(f^\#)}$. A projection ν is said to \triangleright -orient a subset \mathcal{P}' of \mathcal{P} if $\bar{\nu}(s^\#) \triangleright \bar{\nu}(t^\#)$ for all $s^\# \Rightarrow t^\#$ $[\varphi] \in \mathcal{P}'$ and $\bar{\nu}(s^\#) = \bar{\nu}(t^\#)$ for all $s^\# \Rightarrow t^\#$ $[\varphi] \in \mathcal{P} \setminus \mathcal{P}'$. A subterm criterion processor maps $(\mathcal{P}, \mathcal{R})$ to $\{(\mathcal{P} \setminus \mathcal{P}', \mathcal{R})\}$ for some non-empty $\mathcal{P}' \subseteq \mathcal{P}$ which is \triangleright -oriented by some projection for \mathcal{P} .

Choosing $\nu(\text{fold}^\#) = 3$, we have $\bar{\nu}(\text{fold}^\# f y (\text{cons } x l)) = \text{cons } x l \triangleright l = \bar{\nu}(\text{fold}^\# f y l)$, so a subterm criterion processor maps $(\{6\}, \mathcal{R}_{\text{gcd}})$ to $\{(\emptyset, \mathcal{R}_{\text{gcd}})\}$, and $(\emptyset, \mathcal{R}_{\text{gcd}})$ can (trivially) be removed by a graph processor. Now we are left with $(\{3, 4\}, \mathcal{R}_{\text{gcd}})$ and $(\{5\}, \mathcal{R}_{\text{gcd}})$, which involve recursion over integers. We deal with those by means of integer mappings:

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

Integer mapping processors 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 $\varphi \models \bar{\mathcal{J}}(s^\#) \geq 0 \wedge \bar{\mathcal{J}}(s^\#) > \bar{\mathcal{J}}(t^\#)$ for all $s^\# \Rightarrow t^\#$ $[\varphi] \in \mathcal{P}'$, and $\varphi \models \bar{\mathcal{J}}(s^\#) \geq \bar{\mathcal{J}}(t^\#)$ for all $s^\# \Rightarrow t^\#$ $[\varphi] \in \mathcal{P} \setminus \mathcal{P}'$, where $\varphi \models \varphi'$ denotes that $\llbracket \varphi \sigma \rrbracket = 1$ implies $\llbracket \varphi' \sigma \rrbracket = 1$ for each substitution σ which maps variables in $\text{Var}(\varphi) \cup \text{Var}(\varphi')$ to theory values. An integer mapping processor maps $(\mathcal{P}, \mathcal{R})$ to $\{(\mathcal{P} \setminus \mathcal{P}', \mathcal{R})\}$ for some non-empty $\mathcal{P}' \subseteq \mathcal{P}$ which is $>$ -oriented by some integer mapping for \mathcal{P} .

To deal with $(\{5\}, \mathcal{R}_{\text{gcd}})$, let $\mathcal{J}(\text{gcd}^\#)$ be x_2 so $\bar{\mathcal{J}}(\text{gcd}^\# m n) = n$, $\bar{\mathcal{J}}(\text{gcd}^\# n (m \bmod n)) = m \bmod n$ and $m \geq 0 \wedge n > 0 \models n \geq 0 \wedge n > m \bmod n$. Then an integer mapping processor returns $\{(\emptyset, \mathcal{R}_{\text{gcd}})\}$, and $(\emptyset, \mathcal{R}_{\text{gcd}})$ can (trivially) be removed by a graph processor.

Unlike [21], here an integer mapping processor is applicable to $(\{3, 4\}, \mathcal{R}_{\text{gcd}})$: $\mathcal{J}(\text{gcd}^\#) = -x_1$ and the processor returns $\{(\{4\}, \mathcal{R}_{\text{gcd}})\}$. Then $(\{4\}, \mathcal{R}_{\text{gcd}})$ can be removed by a graph processor. This simpler proof is due to the transformation of the call-by-value system; in both SDPs, m and n can be instantiated only to theory values, which is not the case in [21], and a theory argument processor is needed there. To elaborate, we consider the system \mathcal{R} :

$$f \ x \ y \ z \rightarrow f \ x \ (x + 1) \ (x - 1) \ [y < z] \quad c \ x \ y \rightarrow x \quad c \ x \ y \rightarrow y$$

We have $f \ (c \ 0 \ 3) \ 1 \ 2 \rightarrow_{\mathcal{R}} f \ (c \ 0 \ 3) \ ((c \ 0 \ 3) + 1) \ ((c \ 0 \ 3) - 1) \rightarrow_{\mathcal{R}}^+ f \ (c \ 0 \ 3) \ 1 \ 2$, and therefore \mathcal{R} is not terminating with respect to full rewriting. Under call by value, \mathcal{R} is terminating: the first rewrite rule with x appended to the logical constraint gives the only SDP $f^\# \ x \ y \ z \Rightarrow f^\# \ x \ (x + 1) \ (x - 1) \ [y < z, x]$, then let $\mathcal{J}(f^\#)$ be $x_3 - x_2$. Here we benefit from call by value.

5.2 The Chaining of Call-By-Value SDPs

Another benefit of call by value is that we may chain together consecutive SDPs, e.g., $\text{fact}^\# x \Rightarrow u_1^\# x \ 1 \ [t, x]$ and $u_1^\# x \ z \Rightarrow u_2^\# x \ z \ 1 \ [t, x, z]$ may merge to form $\text{fact}^\# x \Rightarrow u_2^\# x \ 1 \ 1 \ [t, x]$. This capability can be important for automatically generated systems.

► **Definition 21.** Given SDPs $p_0 = (s_0^\# \Rightarrow f^\# t'_1 \dots t'_n \ [\varphi_0])$ and $p_1 = (f^\# s'_1 \dots s'_n \Rightarrow t_1^\# \ [\varphi_1])$ where $f \in \mathcal{D}$ and variables are renamed if necessary to avoid name collisions, p_0 and p_1 are called chainable if p_1 is a call-by-value SDP and there exists a substitution σ such that $\text{dom}(\sigma) = \bigcup_{i=1}^n \text{Var}(s'_i)$, $t'_i = s'_i \sigma$ for all i , and $\sigma(x) \in T(\mathcal{F}_\emptyset, \text{Var}(\varphi_0))$ for all $x \in \text{dom}(\sigma) \cap \text{Var}(\varphi_1)$. The chaining $\text{ch}(p_0, p_1)$ of p_0 and p_1 is $s_0^\# \Rightarrow t_1^\# \sigma \ [\varphi_0 \wedge (\varphi_1 \sigma)]$.

Note that in the context of call-by-value rewriting, *all* SDPs can be assumed to be call-by-value, and therefore it is not particularly restrictive to so require p_1 above. To see the importance of this requirement, consider the DP problem $(\{f^\# a b \Rightarrow g^\# (h b a), g^\# (h x y) \Rightarrow f^\# x y\}, \{h b a \rightarrow h a b\})$. If we dropped the call-by-value requirement for the second SDP, the two SDPs would be chainable and yield $f^\# a b \Rightarrow f^\# b a$. However, $f^\# b a$ is not reachable from $f^\# a b$ initially, and replacing the original SDPs with the new one is not sound.

► **Definition 22.** Given a set \mathcal{P} of SDPs and $f \in \mathcal{D}$ such that $\mathcal{P}_f^\ell \neq \emptyset$, $\mathcal{P}_f^r \neq \emptyset$, $\mathcal{P}_f^\ell \cap \mathcal{P}_f^r = \emptyset$ and every pair in $\mathcal{P}_f^r \times \mathcal{P}_f^\ell$ is chainable where $\mathcal{P}_f^\ell = \{s^\# \Rightarrow t^\# [\varphi] \in \mathcal{P} \mid s^\# = f^\# s_1 \cdots s_n\}$ and $\mathcal{P}_f^r = \{s^\# \Rightarrow t^\# [\varphi] \in \mathcal{P} \mid t^\# = f^\# t_1 \cdots t_n\}$, a chaining processor assigns to a DP problem $(\mathcal{P}, \mathcal{R})$ the singleton $\{((\mathcal{P} \setminus (\mathcal{P}_f^\ell \cup \mathcal{P}_f^r)) \cup \mathcal{P}', \mathcal{R})\}$ where $\mathcal{P}' = \{\text{ch}(p_0, p_1) \mid p_0 \in \mathcal{P}_f^r \text{ and } p_1 \in \mathcal{P}_f^\ell\}$.

► **Example 23.** Consider the below set \mathcal{P} of SDPs generated from an imperative program [14]:

$$\begin{array}{llll} \text{fact}^\# x \Rightarrow u_1^\# x \ 1 & [t, x] & u_1^\# x \ z \Rightarrow u_2^\# x \ z \ 1 & [t, x, z] \\ u_2^\# x \ z \ i \Rightarrow u_3^\# x \ z \ i & [i \leq x, z] & u_3^\# x \ z \ i \Rightarrow u_4^\# x \ (z * i) \ i & [t, x, z, i] \\ u_2^\# x \ z \ i \Rightarrow u_5^\# x \ z & [\neg(i \leq x), z] & u_4^\# x \ z \ i \Rightarrow u_2^\# x \ z \ (i + 1) & [t, x, z, i] \end{array}$$

Chaining processors can iteratively remove the occurrences of $u_1^\#$, $u_3^\#$ and $u_4^\#$, and end with (1) $\text{fact}^\# x \Rightarrow u_2^\# x \ 1 \ 1 [t, x]$, (2) $u_2^\# x \ z \ i \Rightarrow u_2^\# x \ (z * i) \ (i + 1) [i \leq x, z]$, and (3) $u_2^\# x \ z \ i \Rightarrow u_5^\# x \ z [\neg(i \leq x), z]$. Note that a chaining processor for $f \in \mathcal{D}$ removes all the occurrences of $f^\#$ in \mathcal{P} , and therefore the process of iteratively applying chaining processors always terminates (on the assumption that $\text{heads}(\mathcal{P})$ is finite). In this example, the above outcome cannot be further chained; there is no chaining processor for u_2 because $\mathcal{P}_{u_2}^\ell \cap \mathcal{P}_{u_2}^r$ contains SDP (2).

► **Example 24.** Consider the following set \mathcal{P} of SDPs:

$$\begin{array}{llll} f^\# x \ y \Rightarrow g^\# (\text{comb } x \ y) & [x \geq 0, y] & g^\# (\text{comb } z \ w) \Rightarrow h^\# (z + w) & [w \geq 0, z] \\ f^\# x \ y \Rightarrow g^\# (\text{comb } (-x) \ y) & [x < 0, y] & g^\# (\text{comb } z \ w) \Rightarrow h^\# (z - w) & [w < 0, z] \end{array}$$

where $\text{comb} : \text{int} \rightarrow \text{int} \rightarrow \text{intpair}$ is a constructor. Note that $\text{comb } z \ w$ is a value. Chaining processors can remove the occurrences of $g^\#$, and end with (1) $f^\# x \ y \Rightarrow h^\# (x + y) [x \geq 0 \wedge y \geq 0]$, (2) $f^\# x \ y \Rightarrow h^\# (x - y) [x \geq 0 \wedge y < 0]$, (3) $f^\# x \ y \Rightarrow h^\# (-x + y) [x < 0 \wedge y \geq 0]$, and (4) $f^\# x \ y \Rightarrow h^\# (-x - y) [x < 0 \wedge y < 0]$.

5.3 Usable Rules

A key processor in the DP framework for full rewriting, which also applies in the innermost setting, is the *reduction pair processor* [21, Definition 25]. This processor is so powerful because it can potentially be used with a wide variety of different reduction pairs and does not require \succ to be monotonic: we must show $s^\# \succ_\varphi t^\#$ or $s^\# \succeq_\varphi t^\#$ for all SDPs $s^\# \Rightarrow t^\# [\varphi]$, and $s \succeq_\varphi t$ for all rules $s \rightarrow t [\varphi]$, and we may then remove the SDPs oriented with \succ .

The challenge is that, no matter how small \mathcal{P} , we must orient *all* rules \mathcal{R} in the DP problem—and all processors considered so far only modify the set \mathcal{P} of SDPs. Fortunately, in the innermost setting, we can see that only some of the rules could potentially be relevant.

To illustrate the idea, consider a system $\mathcal{R}_{\text{drop}}$ which includes *at least* the following rewrite rules and no other defining `drop`, `dfolder`, `cons`, or `nil`:

$$\begin{array}{lll} \text{drop } n \ l \rightarrow l [n \leq 0] & \text{drop } n \ \text{nil} \rightarrow \text{nil} [t, n] & \text{drop } n \ (\text{cons } x \ l) \rightarrow \text{drop } (n - 1) \ l [n > 0] \\ \text{dfolder } f \ y \ n \ \text{nil} \rightarrow y & & [t, n] \\ \text{dfolder } f \ y \ n \ (\text{cons } x \ l) \rightarrow f \ x \ (\text{dfolder } f \ y \ n \ (\text{drop } n \ l)) & & [t, n] \end{array}$$

where $\text{drop} : \text{int} \rightarrow \text{alist} \rightarrow \text{alist}$ and $\text{dfoldr} : (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow \text{int} \rightarrow \text{alist} \rightarrow b$. To deal with the SDP $(\{ \text{dfoldr}^\# f y n (\text{cons } x l) \Rightarrow \text{dfoldr}^\# f y n (\text{drop } n l) [t, n] \}, \mathcal{R}_{\text{drop}})$, we need a reduction pair processor; roughly, we should show that $\text{cons } x l$ is somehow “greater” than $\text{drop } n l$, which cannot be done by a subterm criterion or an integer mapping processor. However, since the variables are to be instantiated to normal forms, any rules other than the ones defining drop would not be used in a chain for this problem. Hence, only these three rules need to be oriented. This observation leads to the notion of usable rules [3, 13]. We base our formulation on a higher-order version [26] and start with two auxiliary definitions:

► **Definition 25.** For $\ell \rightarrow r [\varphi]$ where $\ell : A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$ for $B \in \mathcal{S}$, let $(\ell \rightarrow r [\varphi])^{\text{ex}}$ be $\{ \ell \rightarrow r [\varphi], \ell x_1 \rightarrow r x_1 [\varphi], \dots, \ell x_1 \dots x_n \rightarrow r x_1 \dots x_n [\varphi] \}$ where x_1, \dots, x_n are fresh variables. Let \mathcal{R}^{ex} denote $\bigcup_{\ell \rightarrow r [\varphi] \in \mathcal{R}} (\ell \rightarrow r [\varphi])^{\text{ex}}$.

► **Definition 26.** The set of usable symbols in a term t with respect to a logical constraint φ , denoted by $\mathcal{U}_{\mathcal{F}}(t)[\varphi]$, is defined inductively as follows:

- (1) Suppose $t = f t_1 \dots t_n$ for $f \in \mathcal{F}^\#$. If $t \in T(\mathcal{F}_\emptyset, \text{Var}(\varphi))$, then $\mathcal{U}_{\mathcal{F}}(t)[\varphi] = \emptyset$; otherwise $\mathcal{U}_{\mathcal{F}}(t)[\varphi] = \{ (f, n) \} \cup \bigcup_{i=1}^n \mathcal{U}_{\mathcal{F}}(t_i)[\varphi]$.
- (2) Suppose $t = x t_1 \dots t_n$ for $x \in \mathcal{V}$. If $n = 0$, then $\mathcal{U}_{\mathcal{F}}(t)[\varphi] = \emptyset$; otherwise $\mathcal{U}_{\mathcal{F}}(t)[\varphi] = \{ \perp \}$. Here \perp is a special symbol indicating that potentially any symbol could be usable.

The set of usable symbols for a set \mathcal{P} of SDPs and a set \mathcal{R} of rewrite rules, denoted by $\mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$, is the smallest set U such that (1) $\mathcal{U}_{\mathcal{F}}(t^\#)[\varphi] \subseteq U$ for all $s^\# \Rightarrow t^\# [\varphi] \in \mathcal{P}$, and (2) $\mathcal{U}_{\mathcal{F}}(r)[\varphi] \subseteq U$ for all $(f, n) \in U$ and $f t_1 \dots t_n \rightarrow r [\varphi] \in \mathcal{R}^{\text{ex}}$. Then $\mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$ exists because it is the least pre-fixed point of an order-preserving mapping on a complete lattice.

We present usable rules as a class of DP processors:

► **Definition 27.** Given sets \mathcal{P} of SDPs and \mathcal{R} of rules: if $\perp \notin \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$, the set $\mathcal{U}(\mathcal{P}, \mathcal{R})$ of usable rules is defined as $\{ f t_1 \dots t_k \rightarrow r [\varphi] \in \mathcal{R} \mid (f, n) \in \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R}) \text{ and } k \leq n \}$; otherwise, $\mathcal{U}(\mathcal{P}, \mathcal{R})$ is undefined. A usable-rules processor assigns to a DP problem $(\mathcal{P}, \mathcal{R})$ with $\mathcal{NF}(\mathcal{Q}) \subseteq \mathcal{NF}(\mathcal{R})$ the singleton $\{ (\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R})) \}$.

► **Example 28.** We continue the discussion about drop and dfoldr . Consider the DP problem $(\mathcal{P}, \mathcal{R}_{\text{drop}})$ where $\mathcal{P} = \{ \text{dfoldr}^\# f y n (\text{cons } x l) \Rightarrow \text{dfoldr}^\# f y n (\text{drop } n l) [t, n] \}$. We have $\mathcal{U}_{\mathcal{F}}(\text{dfoldr}^\# f y n (\text{drop } n l) [t, n]) = \{ (\text{dfoldr}^\#, 4), (\text{drop}, 2) \}$ and can derive $\mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R}) = \{ (\text{dfoldr}^\#, 4), (\text{drop}, 2), (\text{nil}, 0) \}$. Hence, $\mathcal{U}(\mathcal{P}, \mathcal{R})$ consists of the three rules defining drop .

5.4 Argument Filterings

Let us consider the role of \perp in the definition of usable rules: $\mathcal{U}(\mathcal{P}, \mathcal{R})$ is undefined if $\perp \in \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$ —which occurs if the right-hand side of some SDP in \mathcal{P} , or the right-hand side of any rule $f t_1 \dots t_n \rightarrow r [\varphi] \in \mathcal{R}^{\text{ex}}$ with $(f, n) \in \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$, is not a pattern. This is not a rare occasion in higher-order rewriting. For example, let us rework dfoldr into dfoldl :

$$\text{dfoldl } f y n \text{ nil} \rightarrow y [t, n] \quad \text{dfoldl } f y n (\text{cons } x l) \rightarrow \text{dfoldl } f (f y x) n (\text{drop } n l) [t, n]$$

where drop is defined by the same rules as before. Now we have $\text{dfoldl}^\# f y n (\text{cons } x l) \Rightarrow \text{dfoldl}^\# f (f y x) n (\text{drop } n l) [t, n]$, which produces \perp .

However, observe that the problematic subterm $f y x$ and the decreasing subterm $\text{drop } n l$ occur in different arguments of $\text{dfoldl}^\#$. If we use a reduction pair that considers only the fourth argument, intuitively we can disregard any rules that may be used to reduce the second. This is formalized via an *argument filtering* [3, 26], which temporarily removes

certain subterms from both sides of an ordering requirement before computing usable rules, often drastically reducing the number of usable rules that the reduction pair must consider.

Traditionally, argument filterings are defined for function symbols with a fixed number of arguments. Extending this notion to our curried setting imposes new technical challenges; e.g., if we filter away the second argument of $f : A \rightarrow B \rightarrow C$, what type to use for the filtering of $f\ x, C$ or $B \rightarrow C$? Fortunately, this problem is solvable if we do not allow variables to occur at the head of an application in the result of a filtering. This is very different from the approach in [2], which heavily restricts what filtering can be used, to ensure that a variable of higher type and all its possible instances have the same filtering applied to them.

► **Definition 29.** We assume given a set \mathcal{S}_1 of sorts such that $\mathcal{S}_\vartheta \subseteq \mathcal{S}_1$, and a mapping μ from \mathcal{T} to \mathcal{S}_1 such that $\mu(A) = A$ for all $A \in \mathcal{S}_\vartheta$. In addition, for each function symbol $f : A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$ with $B \in \mathcal{S}$, we assume given a set $\text{regard}(f) \subseteq \{1, \dots, n\}$. Define:

$$\begin{aligned} \Sigma = \mathcal{F}_\vartheta \cup \{ & f_m : \mu(A_{i_1}) \rightarrow \dots \rightarrow \mu(A_{i_k}) \rightarrow \mu(A_{m+1} \rightarrow \dots \rightarrow A_n \rightarrow B) \mid \\ & f : A_1 \rightarrow \dots \rightarrow A_n \rightarrow B \in \mathcal{F}^\# \text{ with } B \in \mathcal{S} \text{ and } 0 \leq m \leq n \text{ and} \\ & \text{regard}(f) \cap \{1, \dots, m\} = \{i_1, \dots, i_k\} \text{ and } i_1 < \dots < i_k \} \\ & \cup \{ \bullet_{\mu(A)} : \mu(A) \mid A \in \mathcal{T} \} \end{aligned}$$

The argument filtering with respect to a logical constraint φ is a mapping π_φ from $T(\mathcal{F}^\#, \mathcal{V})$ to $T(\Sigma^\#, \{x : \mu(A) \mid x : A \in \mathcal{V}\})$, defined as follows:

- (1) $\pi_\varphi(f\ t_1 \dots t_n) = f\ t_1 \dots t_n$ if $f\ t_1 \dots t_n \in T(\mathcal{F}_\vartheta, \text{Var}(\varphi))$ and has a base type
- (2) $\pi_\varphi(f\ t_1 \dots t_n) = f_n\ \pi_\varphi(t_{i_1}) \dots \pi_\varphi(t_{i_k})$ otherwise, where $\{i_1, \dots, i_k\} = \text{regard}(f) \cap \{1, \dots, n\}$ and $i_1 < \dots < i_k$
- (3) $\pi_\varphi(x) = x$ for $x \in \mathcal{V}$; and $\pi_\varphi(x\ t_1 \dots t_n) = \bullet_{\mu(A)}$ if $x \in \mathcal{V}$ and $n > 0$

By definition, $\pi_\varphi(t) : \mu(A)$ for all $t : A$. Moreover, no subterm of $\pi_\varphi(t)$ has a variable at the head of an application, and all function symbols have a first-order type and occur maximally applied. Hence, $\pi_\varphi(t)$ is a (many-sorted) first-order term, just written in applicative notation. For \mathcal{S}_1 , we may for instance choose $\mathcal{S}_\vartheta \cup \{\mathbf{o}\}$, i.e., there is a single sort besides theory sorts.

Now we define usable rules with respect to an argument filtering:

► **Definition 30.** Given the mapping $\text{regard}(\cdot)$, $\mathcal{U}_\mathcal{F}(f\ t_1 \dots t_n)[\varphi]$ is redefined as $\{(f, n)\} \cup \bigcup_{i \in \text{regard}(f) \cap \{1, \dots, n\}} \mathcal{U}_\mathcal{F}(t_i)[\varphi]$ in the case where $f\ t_1 \dots t_n \notin T(\mathcal{F}_\vartheta, \text{Var}(\varphi))$. The definition in other cases is unchanged, and so is the definition of $\mathcal{U}_\mathcal{F}(\mathcal{P}, \mathcal{R})$.

Let U_1 be $\{\pi_\varphi(f\ t_1 \dots t_n) \rightarrow \pi_\varphi(r) [\varphi] \mid (f, n) \in \mathcal{U}_\mathcal{F}(\mathcal{P}, \mathcal{R}) \text{ and } f\ t_1 \dots t_n \rightarrow r [\varphi] \in \mathcal{R}^{\text{ex}}\}$, and U_2 be $\{f_n\ x_{i_1} \dots x_{i_k} \rightarrow y \mid [y = f\ x_1 \dots x_n] \mid (f, n) \in \mathcal{U}_\mathcal{F}(\mathcal{P}, \mathcal{R}), f \in \mathcal{F}_\vartheta, n > 0, f : A_1 \rightarrow \dots \rightarrow A_n \rightarrow B \text{ with } B \in \mathcal{S}, \text{regard}(f) = \{i_1, \dots, i_k\} \text{ and } i_1 < \dots < i_k\}$.

We redefine $\mathcal{U}(\mathcal{P}, \mathcal{R})$ as $U_1 \cup U_2$ if $\perp \notin \mathcal{U}_\mathcal{F}(\mathcal{P}, \mathcal{R})$, and $p + i \in \text{regard}(f)$ for all $(f, n) \in \mathcal{U}_\mathcal{F}(\mathcal{P}, \mathcal{R})$, $f\ s_1 \dots s_p \rightarrow g\ t_1 \dots t_q [\varphi] \in \mathcal{R}$ where $p < n$ and $g \in \mathcal{F}$, and $i \in \{1, \dots, n - p\}$ such that $q + i \in \text{regard}(g)$. Otherwise, $\mathcal{U}(\mathcal{P}, \mathcal{R})$ is undefined.

That is, we consider usable rules only with respect to the positions that are not filtered away. The requirement that $p + i \in \text{regard}(f)$ if (f, n) is a usable symbol and $p + i \leq n$ is a technical limitation needed to ensure that applying a rewrite rule at the head of an application does not conflict with the argument filtering.

Next, we recall the notion of a constrained reduction pair from [21], but use a more liberal monotonicity requirement due to the first-order setting created by the filtering.

► **Definition 31.** A constrained relation R is a set of triples (s, t, φ) where s and t are (first-order) terms which have the same sort and φ is a logical constraint. We write $s R_\varphi t$

if $(s, t, \varphi) \in R$. A binary relation R' over terms is said to cover a constrained relation R if $s R_\varphi t$ implies that $(s\sigma) \downarrow_\kappa R' (t\sigma) \downarrow_\kappa$ for each substitution σ which respects φ .

A constrained reduction pair (\succeq, \succ) is a pair of constrained relations where \succeq is covered by some reflexive and transitive relation \sqsupseteq such that $t_k \sqsupseteq t'_k$ implies $f t_1 \cdots t_{k-1} t_k t_{k+1} \cdots t_n \sqsupseteq f t_1 \cdots t_{k-1} t'_k t_{k+1} \cdots t_n$ (i.e., monotonicity) for all $f \in \Sigma$ and $k \in \{1, \dots, n\}$, \succ is covered by some well-founded relation \sqsubset , and $\sqsubset; \sqsupseteq \subseteq \sqsupseteq^+$.

Having this, we can define reduction pair processors with respect to an argument filtering:

► **Definition 32.** A reduction pair processor with argument filterings assigns to a DP problem $(\mathcal{P}, \mathcal{R})$ with $\mathcal{NF}(\mathcal{Q}) \subseteq \mathcal{NF}(\mathcal{R})$ the singleton $\{(\mathcal{P} \setminus \mathcal{P}', \mathcal{R})\}$ for some non-empty $\mathcal{P}' \subseteq \mathcal{P}$ if there exists a mapping $\text{regard}(\cdot)$ and a constrained reduction pair (\succeq, \succ) such that (1) $\pi_\varphi(s^\#) \succ_\varphi \pi_\varphi(t^\#)$ for all $s^\# \Rightarrow t^\# [\varphi] \in \mathcal{P}'$, (2) $\pi_\varphi(s^\#) \succeq_\varphi \pi_\varphi(t^\#)$ for all $s^\# \Rightarrow t^\# [\varphi] \in \mathcal{P} \setminus \mathcal{P}'$, and (3) $\ell \succeq_\varphi r$ for all $\ell \rightarrow r [\varphi] \in \mathcal{U}(\mathcal{P}, \mathcal{R})$.

► **Example 33.** Consider dfoldl , with the following DP problem: $(\{\text{dfoldl}^\# f y n (\text{cons } x l) \Rightarrow \text{dfoldl}^\# f (f y x) n (\text{drop } n l) [t, n]\}, \mathcal{R})$. Let $\text{regard}(\text{dfoldl}^\#)$ be $\{4\}$, $\text{regard}(\text{drop})$ be $\{2\}$, and $\text{regard}(\text{cons})$ be $\{2\}$. We are obliged to prove (1) $\text{dfoldl}_4^\# (\text{cons}_2 l) \succ_{t,n} \text{dfoldl}_4^\# (\text{drop}_2 l)$, (2) $\text{drop}_2 l \succeq_{n \leq 0} l$, (3) $\text{drop}_2 \text{nil}_0 \succeq_{t,n} \text{nil}_0$, and (4) $\text{drop}_2 (\text{cons}_2 l) \succeq_{n > 0} \text{drop}_2 l$. The recursive path ordering for first-order LCTRSs [29] can be used to fulfill these obligations.

6 Universal Computability with Usable Rules

In this section, we study universal computability [21] with respect to innermost rewriting. This concept concerns a modular programming scenario where the LCSTRS represents a single module in a larger program, and corresponds to the termination of a function in all “reasonable” uses, including unknown uses. We recall the notion of a hierarchical combination [31, 32, 33, 9] rephrased in terms of LCSTRSs:

► **Definition 34** ([21]). 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; one may think of \mathcal{R}_0 as an imported module.

Universal computability is defined on the basis of hierarchical combinations:

► **Definition 35** ([21]). 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' . \mathcal{R}_0 is called universally computable if all its terms are.

In summary, we consider passing \mathbb{C} -computable arguments to a defined symbol in \mathcal{R}_0 the “reasonable” way of calling the function. We use SDPs to establish universal computability:

► **Theorem 36.** An accessible function passing system \mathcal{R}_0 with sort ordering \succsim is universally computable if there exists no infinite innermost $(\text{SDP}(\mathcal{R}_0), \mathcal{R}_0 \cup \mathcal{R}_1)$ -chain for any extension \mathcal{R}_1 of \mathcal{R}_0 and extension \succsim' of \succsim to sorts in $\mathcal{R}_0 \cup \mathcal{R}_1$.

This is a more general version of Theorem 14 and will be proved alongside it in Appendix A.2. Note that the original set of rules, \mathcal{Q} , in the definition of innermost chain is now $\mathcal{R}_0 \cup \mathcal{R}_1$. We modify the DP framework to prove universal computability for a fixed base system \mathcal{R}_0 .

► **Definition 37.** A (universal) DP problem $(\mathcal{P}, \mathcal{R}, \mathbf{p})$ consists of a set \mathcal{P} of SDPs, a set \mathcal{R} of rewrite rules and a flag $\mathbf{p} \in \{\mathbf{def}, \mathbf{ind}\}$ (for definite or indefinite). A DP problem $(\mathcal{P}, \mathcal{R}, \mathbf{p})$ is finite if either (1) $\mathbf{p} = \mathbf{def}$ and there exists no infinite innermost $(\mathcal{P}, \mathcal{R})$ -chain (with $\mathcal{Q} := \mathcal{R}_0$), or (2) $\mathbf{p} = \mathbf{ind}$ and there exists no infinite innermost $(\mathcal{P}, \mathcal{R} \cup \mathcal{R}_1)$ -chain for any extension \mathcal{R}_1 of \mathcal{R}_0 (with $\mathcal{Q} := \mathcal{R}_0 \cup \mathcal{R}_1$).

DP processors are defined in the same way as before, now for universal DP problems. The goal is to show that $(\text{SDP}(\mathcal{R}_0), \mathcal{R}_0, \mathbf{ind})$ is finite, and the procedure for termination in Section 5 still works if we change the initialization accordingly. Unlike [21], here we have the advantage of usable rules: if $\mathcal{U}(\mathcal{P}, \mathcal{R})$ is defined, then $\mathcal{U}(\mathcal{P}, \mathcal{R} \cup \mathcal{R}_1) = \mathcal{U}(\mathcal{P}, \mathcal{R})$ for any extension \mathcal{R}_1 (provided all symbols in \mathcal{P} and \mathcal{R} are from \mathcal{R}_0 , which is typically the case as DP processors generally do not introduce new symbols). Hence a usable-rules processor assigns to a DP problem $(\mathcal{P}, \mathcal{R}, \mathbf{p})$ the singleton $\{(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}), \mathbf{def})\}$.

Even if usable-rules processors are not applicable, we may still use a reduction pair processor with an argument filtering: we do not have to orient the rules in \mathcal{R}_1 since they cannot be usable. Referring to Definition 32, a reduction pair processor now assigns to a DP problem $(\mathcal{P}, \mathcal{R}, \mathbf{p})$ the singleton $\{(\mathcal{P} \setminus \mathcal{P}', \mathcal{R}, \mathbf{p})\}$ without changing the input flag because it does not permanently discard any rule. The other DP processors discussed in Section 5 apply to universal DP problems similarly; they just keep the input flag unchanged in the output.

7 Implementation and Evaluation

We have implemented our results in Cora [27], using a version of HORPO (based on the initial definition in [22]) and its first-order limitation RPO as the only reduction pair. Constraint validity checks are delegated to the SMT solver Z3 [8]. We also use Z3 to seek a good argument filtering and HORPO instance by encoding the requirements into an SMT problem.

Our implementation of the chaining processor from Definition 22 aims to minimize the number of DPs in the resulting problem. To this end, it chooses a function symbol f such that the expression $|\mathcal{P}'| - |\mathcal{P}_f^\ell \cup \mathcal{P}_f^r|$ with the sets defined as in Definition 22 is minimized.

We evaluated Cora on three groups of benchmarks: our own collected LCSTRS benchmarks, the lambda-free problems from the higher-order category of the TPDB [7], and problems from the first-order “integer TRS innermost” category. The results are summarized below, where “full” considers the framework for full termination with the methods from [21], and “call-by-value” does the transformation of Lemma 7 before applying the innermost framework:

	Termination			Universal Computability		
	Full	Innermost	Call-by-value	Full	Innermost	Call-by-value
Total yes	171	179	182	155	179	182
Total maybe	104	96	93	116	96	93

Note that we gain significant power compared to [21] when analyzing universal computability. This is largely due to the two usable-rules processors: the reduction pair processor cannot be applied on its own in an indefinite DP problem, so either changing the flag from \mathbf{ind} to \mathbf{def} , or being able to omit the extra rules in a reduction pair, gives a lot of power.

However, the gains for *termination* are more modest. We suspect the reason is that the new processors primarily find their power in large systems (where it is essential to eliminate many rules when searching for a reduction pair), and automatically generated systems (where there are often many chainable rules), not in the handcrafted benchmarks of the TPDB.

Another observation is that there is no difference in proving power between termination or universal computability, both for innermost and for call-by-value reduction. This may be

due to Cora having only one reduction pair (HORPO): on this benchmark set, when HORPO can be applied to simplify a DP problem, then it is also possible to filter away problematic subterms that would stop us from applying usable rules.

A detailed evaluation page is available through the following link:

<https://www.cs.ru.nl/~cynthiakop/experiments/fscd25/>

Comparison to Other Analyzers. While we have made progress, the additions of this paper are not yet sufficient for Cora to compete with dedicated analyzers for first-order integer rewriting, or unconstrained higher-order term rewriting.

In the “integer TRS innermost” category, AProVE [15] can prove innermost termination of 102 benchmarks, while Cora can handle 72 for innermost evaluation and 73 for call-by-value. The most important difference seems to be that AProVE has a much more sophisticated implementation of what we call the integer mapping processor as a reduction pair processor with polynomial interpretations and usable rules with respect to argument filterings [12]. In addition, these benchmarks often have rules such as $f(x) \rightarrow g(x > 0, x)$, $g(t) \rightarrow r_1$, $g(f) \rightarrow r_2$, which would benefit from a transformation to turn the Boolean subterm $x > 0$ into a proper constraint. Improving on these points is obvious future work.

In the “higher-order union beta” category, Wanda [25] can prove termination of full rewriting of 105 benchmarks, while Cora can handle 79 for innermost or call-by-value reduction. Since these benchmarks are unconstrained, Cora does not benefit here from any of the processors that consider the theory, while Wanda does have many more features to increase its prover ability; for example, polynomial interpretations, dynamic dependency pairs, and delegation of partial problems to a first-order termination tool.

8 Related Work

Dependency Pair frameworks are the cornerstone of most fully automated termination analysis tools for various flavors of first-order [1, 19, 23] and higher-order [2, 13, 30] rewriting. A common theme of these DP frameworks lies in the notions of a problem to be analyzed for presence of infinite chains and processors to advance the proofs of (non-)termination. The notion of DP frameworks has been lifted to *constrained* rewriting, both in the first-order [10, 12] and recently also in the higher-order [21] setting. The latter work also includes the notion of universal computability, allowing for termination proofs also in the presence of further library functions that are not known at the time of analysis.

Our present DP framework for innermost rewriting is specially designed with termination analysis of functional programs in call-by-value languages (e.g., Scala, OCaml) in mind as a target application. Such LCSTRSs may come from an automated syntactic translation and thus have more rewrite rules with “intermediate” defined symbols than hand-optimized rewrite systems. Our *chaining processor* renders the analysis of such problems feasible. This kind of program transformation is used also in other program analysis tools [4, 10, 14, 15]; here, we have adapted the technique to higher-order rewriting with constraints.

Usable rules w.r.t. an argument filtering were introduced for first-order rewriting in [19] and soon extended to simply-typed higher-order rewriting for arbitrary rewrite strategies in [2]. Our contribution here is to lift usable rules w.r.t. an argument filtering to a setting with theory constraints and to universal computability, thus opening up the door for applications in analysis of programs for real-world languages. Moreover, we adapt the technique to innermost rewriting and thus do not require that the constrained reduction pair (\succeq, \succ) satisfies $c \ x \ y \succeq x$ and $c \ x \ y \succeq y$ for fresh symbols c .

9 Conclusion and Future Work

In this paper, we have extended the static dependency pair framework for LCSTRSs to *innermost* and *call-by-value* evaluation strategies. In doing so, we have adapted several existing processors for full termination and proposed three processors—chaining, usable rules, reduction pairs with usable rules w.r.t. argument filterings—that were not present for the setting of full termination. These processors apply not only to conventional closed-world termination analysis, but also to *open-world* termination analysis via universal computability, where the presence of further rewrite rules for library functions is assumed, thus broadening the set of potential start terms that must be considered. Our experimental results on several benchmark collections indicate improvements over the state of the art by exploiting the evaluation strategy via the new processors, most pronounced for universal computability.

There are several directions for future work. Our *chaining processors* could be improved, e.g., by using unification instead of matching, thus defining a form of *narrowing* for our constrained DPs in the higher-order setting. Moreover, so far our implementation prohibits chaining a DP with itself to prevent non-termination of repeated applications of the chaining processors. We might loosen this restriction via appropriate heuristics to detect cases in which such self-chaining could be beneficial. In addition, we could investigate chaining not just for DPs, but also for rewrite rules. The integer mapping processor could be improved by lifting *polynomial interpretations* [43] for higher-order rewriting to the constrained setting of LCSTRSs. Another improvement would consist of moving Boolean expressions from the right-hand side of rewrite rules into their constraints. Techniques like the integer mapping processor or the subterm criterion, which establish weak or strict decrease between certain arguments of dependency pairs, could also be improved by combining them with the *size-change termination* principle [37]. Size-change termination has been integrated into the first-order DP framework [42, 5], and a natural next step would be to lift this integration to the higher-order DP framework for LCSTRSs, both for innermost and for full termination.

A different avenue could be to research to what extent the termination analysis techniques in this paper can be ported from innermost or call-by-value evaluation to *arbitrary* evaluation strategies, or to *lazy evaluation*, as used in Haskell. For the latter, we might consider an approximation of lazy evaluation via a higher-order form of context-sensitive rewriting [1, 38].

Finally, to go back to one of the main motivations of this work, we could devise translations from higher-order functional programming languages with call-by-value semantics (e.g., Scala, OCaml) to LCSTRSs. This would allow for applying the contributions of this paper to prove termination of programs written in these languages, a long-term goal of this line of research.

References

- 1 B. Alarcón, R. Gutiérrez, and S. Lucas. Context-sensitive dependency pairs. *IC*, 208(8):922–968, 2010. doi:10.1016/J.IC.2010.03.003.
- 2 T. Aoto and T. Yamada. Argument filterings and usable rules for simply typed dependency pairs. In S. Ghilardi and R. Sebastiani, editors, *Proc. FroCoS*, pages 117–132, 2009. doi:10.1007/978-3-642-04222-5_7.
- 3 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.
- 4 D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In A. Biere and C. Pixley, editors, *Proc. FMCAD*, pages 25–32, 2009. doi:10.1109/FMCAD.2009.5351147.

- 5 M. Codish, C. Fuhs, J. Giesl, and P. Schneider-Kamp. Lazy abstraction for size-change termination. In C. G. Fermüller and A. Voronkov, editors, *Proc. LPAR (Yogyakarta)*, pages 217–232, 2010. doi:10.1007/978-3-642-16242-8_16.
- 6 Community. Termination competition (TermCOMP). URL: https://termination-portal.org/wiki/Termination_Competition.
- 7 Community. The termination problem database (TPDB). URL: <https://github.com/TermCOMP/TPDB>.
- 8 L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C.R. Ramakrishnan and J. Rehof, editors, *Proc. TACAS*, pages 337–340, 2008. doi:10.1007/978-3-540-78800-3_24.
- 9 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.
- 10 S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In R. A. Schmidt, editor, *Proc. CADE*, pages 277–293, 2009. doi:10.1007/978-3-642-02959-2_22.
- 11 S. Falke, D. Kapur, and C. Sinz. Termination analysis of C programs using compiler intermediate languages. In M. Schmidt-Schauß, editor, *Proc. RTA*, pages 41–50, 2011. doi:10.4230/LIPIcs.RTA.2011.41.
- 12 C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In R. Treinen, editor, *Proc. RTA*, pages 32–47, 2009. doi:10.1007/978-3-642-02348-4_3.
- 13 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.
- 14 C. Fuhs, C. Kop, and N. Nishida. Verifying procedural programs via constrained rewriting induction. *ACM TOCL*, 18(2):14:1–14:50, 2017. doi:10.1145/3060143.
- 15 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(1):3–31, 2017. doi:10.1007/s10817-016-9388-y.
- 16 J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM TOPLAS*, 33(2):7:1–7:39, 2011. doi:10.1145/1890028.1890030.
- 17 J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In A. King, editor, *Proc. PPDP*, pages 1–12, 2012. doi:10.1145/2370776.2370778.
- 18 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.
- 19 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *JAR*, 37(3):155–203, 2006. doi:10.1007/s10817-006-9057-7.
- 20 L. Guo, K. Hagens, C. Kop, and D. Vale. Higher-order constrained dependency pairs for (universal) computability. pre-publication Arxiv copy of [21] with additional appendix. doi: <https://doi.org/10.48550/arXiv.2406.19379>.
- 21 L. Guo, K. Hagens, C. Kop, and D. Vale. Higher-order constrained dependency pairs for (universal) computability. In R. Kráľovič and A. Kučera, editors, *Proc. MFCS*, pages 57:1–57:15, 2024. doi:10.4230/LIPIcs.MFCS.2024.57.
- 22 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.
- 23 N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *IC*, 199(1-2):172–199, 2005. doi:10.1016/J.IC.2004.10.004.
- 24 N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: techniques and features. *IC*, 205(4):474–511, 2007. doi:10.1016/j.ic.2006.08.010.

- 25 C. Kop. WANDA – a higher order termination tool (system description). In Z. M. Ariola, editor, *Proc. FSCD*, pages 36:1–36:19, 2020. doi:10.4230/LIPICS.FSCD.2020.36.
- 26 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.
- 27 C. Kop et al. The Cora analyzer. URL: <https://github.com/hezzel/cora>.
- 28 C. Kop et al. hezzel/cora: FSCD 2025. doi:10.5281/zenodo.15318964.
- 29 C. Kop and N. Nishida. Term rewriting with logical constraints. In P. Fontaine, C. Ringeisen, and R. A. Schmidt, editors, *Proc. FroCoS*, pages 343–358, 2013. doi:10.1007/978-3-642-40885-4_24.
- 30 C. Kop and F. van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *LMCS*, 8(2), 2012. doi:10.2168/LMCS-8(2:10)2012.
- 31 M. R. K. Krishna 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.
- 32 M. R. K. Krishna 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.
- 33 M. R. K. Krishna 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.
- 34 K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Trans. Inf. Syst.*, E92.D(10):2007–2015, 2009. doi:10.1587/transinf.E92.D.2007.
- 35 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.
- 36 K. Kusakari and M. Sakai. Static dependency pair method for simply-typed term rewriting and related techniques. *IEICE Trans. Inf. Syst.*, E92.D(2):235–247, 2009. doi:10.1587/transinf.E92.D.235.
- 37 C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In C. Hankin and D. Schmidt, editors, *Proc. POPL*, pages 81–92, 2001. doi:10.1145/360204.360210.
- 38 S. Lucas. Context-sensitive computations in functional and functional logic programs. *JFLP*, 1998(1), 1998.
- 39 C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java bytecode by term rewriting. In C. Lynch, editor, *Proc. RTA*, pages 259–276, 2010. doi:10.4230/LIPICS.RTA.2010.259.
- 40 S. Suzuki, K. Kusakari, and F. Blanqui. Argument filterings and usable rules in higher-order rewrite systems. *IPSJ Online Trans.*, 4:114–125, 2011. doi:10.2197/ipsjtrans.4.114.
- 41 R. Thiemann. *The DP framework for proving termination of term rewriting*. PhD thesis, RWTH Aachen University, Germany, 2007. URL: <http://darwin.bth.rwth-aachen.de/opus3/volltexte/2007/2066/>.
- 42 R. Thiemann and J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. *AAECC*, 16(4):229–270, 2005. doi:10.1007/S00200-005-0179-7.
- 43 J. van de Pol. Termination proofs for higher-order rewrite systems. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Proc. HOA*, pages 305–325, 1993. doi:10.1007/3-540-58233-9_14.
- 44 F. van Raamsdonk. Translating logic programs into conditional rewriting systems. In L. Naish, editor, *Proc. ICLP*, pages 168–182, 1997. doi:10.7551/mitpress/4299.003.0018.
- 45 H. Zantema. Termination of term rewriting by semantic labelling. *FI*, 24(1/2):89–105, 1995. doi:10.3233/FI-1995-24124.

A Full Proofs

A.1 Soundness for Section 5

For soundness of the graph, subterm criterion and integer mapping processors, we refer to Appendix A.2 in [20]; only minimal changes are needed. Below we address the rest.

► **Theorem 38.** *Chaining processors are sound.*

Proof of Theorem 38. Given a DP problem $(\mathcal{P}, \mathcal{R})$, a defined symbol f with the said properties and an infinite innermost $(\mathcal{P}, \mathcal{R})$ -chain $(s_0^\# \Rightarrow t_0^\# [\varphi_0], \sigma_0), (s_1^\# \Rightarrow t_1^\# [\varphi_1], \sigma_1), \dots$ where variables are renamed to avoid name collisions between any two of the SDPs and $\text{dom}(\sigma_i) \subseteq \text{Var}(s_i^\#) \cup \text{Var}(t_i^\#) \cup \text{Var}(\varphi_i)$ for all i , we consider $k > 0$ such that $t_{k-1}^\# = f^\# t'_1 \dots t'_n$ and $s_k^\# = f^\# s'_1 \dots s'_n$. By definition, $t'_i \sigma_{k-1} \xrightarrow{\mathcal{R}}^* s'_i \sigma_k$ for all i . Because $s_{k-1}^\# \Rightarrow t_{k-1}^\# [\varphi_{k-1}]$ and $s_k^\# \Rightarrow t_k^\# [\varphi_k]$ are chainable, there exists a substitution σ' such that $\text{dom}(\sigma') = \bigcup_{i=1}^n \text{Var}(s'_i)$, $s'_i \sigma' = t'_i$ for all i and $\sigma'(x) \in T(\mathcal{F}_\vartheta, \text{Var}(\varphi_{k-1}))$ for all $x \in \text{dom}(\sigma') \cap \text{Var}(\varphi_k)$. Hence, $s'_i \sigma' \sigma_{k-1} \xrightarrow{\mathcal{R}}^* s'_i \sigma_k$ for all i . Since s'_i is a value for all i , $\sigma_{k-1}(\sigma'(x)) \xrightarrow{\mathcal{R}}^* \sigma_k(x)$ for all $x \in \bigcup_{i=1}^n \text{Var}(s'_i)$. Now we show that replacing $(s_{k-1}^\# \Rightarrow t_{k-1}^\# [\varphi_{k-1}], \sigma_{k-1})$ and $(s_k^\# \Rightarrow t_k^\# [\varphi_k], \sigma_k)$ by $(s_{k-1}^\# \Rightarrow t_k^\# \sigma' [\varphi_{k-1} \wedge (\varphi_k \sigma')], \sigma_{k-1} \cup \sigma_k)$ yields an infinite innermost chain. First, it is routine to verify that $\sigma_{k-1} \cup \sigma_k$ respects $\varphi_{k-1} \wedge (\varphi_k \sigma')$. Next, $s_{k-1}^\# (\sigma_{k-1} \cup \sigma_k) = s_{k-1}^\# \sigma_{k-1}$ so it is in normal form. Last, $t_k^\# \sigma' (\sigma_{k-1} \cup \sigma_k) \xrightarrow{\mathcal{R}}^* t_k^\# \sigma_k \xrightarrow{\mathcal{R}}^* s_{k+1}^\# \sigma_{k+1}$. ◀

We present some auxiliary results for the two classes of DP processors based on usable rules in a unified way, regardless of whether an argument filtering is applied. When no filtering is applied, let $\text{regard}(f) = \{1, \dots, n\}$ for each $f : A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$ with $B \in \mathcal{S}$.

► **Definition 39.** *The set $\mathcal{U}_\mathcal{F}^\alpha(t)$ of actually usable symbols in a term t is (1) $\{(f, n)\} \cup \bigcup_{i \in \text{regard}(f) \cap \{1, \dots, n\}} \mathcal{U}_\mathcal{F}^\alpha(t_i)$ if $t = f t_1 \dots t_n$ for $f \in \mathcal{F}^\#$, t is not a ground theory term, and is not in normal form (with respect to \mathcal{Q}), or (2) \emptyset otherwise.*

► **Lemma 40.** *Given a constraint φ and a substitution σ such that $\sigma(x)$ is in normal form for all x and is a theory value if $x \in \text{Var}(\varphi)$: $\mathcal{U}_\mathcal{F}^\alpha(t\sigma) \subseteq \mathcal{U}_\mathcal{F}(t)[\varphi]$ for all t with $\perp \notin \mathcal{U}_\mathcal{F}(t)[\varphi]$.*

Proof of Lemma 40. By induction on t . If $t = x t_1 \dots t_n$ with $x \in \mathcal{V}$, then $n = 0$ because $\perp \notin \mathcal{U}_\mathcal{F}(t)[\varphi]$. Hence, $t\sigma = \sigma(x)$ is in normal form, and therefore $\mathcal{U}_\mathcal{F}^\alpha(t\sigma) = \emptyset$. Otherwise, $t = f t_1 \dots t_n$ for $f \in \mathcal{F}^\#$. If $t\sigma = f(t_1\sigma) \dots (t_n\sigma)$ is not a ground theory term, $t \notin T(\mathcal{F}_\vartheta, \text{Var}(\varphi))$ because $\sigma(x)$ is a theory value for all $x \in \text{Var}(\varphi)$. So if $\mathcal{U}_\mathcal{F}^\alpha(t\sigma) \neq \emptyset$, $\mathcal{U}_\mathcal{F}(t)[\varphi] = \{(f, n)\} \cup \bigcup_{i \in \text{regard}(f) \cap \{1, \dots, n\}} \mathcal{U}_\mathcal{F}(t_i)[\varphi]$, and therefore $\perp \notin \mathcal{U}_\mathcal{F}(t_i)[\varphi]$ for each such i . By induction, $\mathcal{U}_\mathcal{F}^\alpha(t_i\sigma) \subseteq \mathcal{U}_\mathcal{F}(t_i)[\varphi]$. Hence, $\mathcal{U}_\mathcal{F}^\alpha(t\sigma) \subseteq \mathcal{U}_\mathcal{F}(t)[\varphi]$. ◀

► **Lemma 41.** *Given a set \mathcal{P} of SDPs and a set \mathcal{R} of rules such that $\mathcal{NF}(\mathcal{Q}) \subseteq \mathcal{NF}(\mathcal{R})$ and $\mathcal{U}(\mathcal{P}, \mathcal{R})$ is defined: for all terms t, t' , if $\mathcal{U}_\mathcal{F}^\alpha(t) \subseteq \mathcal{U}_\mathcal{F}(\mathcal{P}, \mathcal{R})$ and $t \xrightarrow{\mathcal{R}} t'$, $\mathcal{U}_\mathcal{F}^\alpha(t') \subseteq \mathcal{U}_\mathcal{F}(\mathcal{P}, \mathcal{R})$.*

Proof of Lemma 41. By induction on t .

(1) $t = f t_1 \dots t_n$ for $f \in \mathcal{F}$, and there exist a rewrite rule $\ell \rightarrow r$ $[\varphi] \in \mathcal{R}$, substitution σ and number $p \leq n$ such that $f t_1 \dots t_p = \ell\sigma$, $t' = (r\sigma) t_{p+1} \dots t_n$, $\sigma(x)$ is in normal form for all x and σ respects φ . Since $\mathcal{NF}(\mathcal{Q}) \subseteq \mathcal{NF}(\mathcal{R})$, t is not in \mathcal{Q} -normal form, so $(f, n) \in \mathcal{U}_\mathcal{F}^\alpha(t) \subseteq \mathcal{U}_\mathcal{F}(\mathcal{P}, \mathcal{R})$, and therefore $\mathcal{U}_\mathcal{F}(r x_{p+1} \dots x_n)[\varphi] \subseteq \mathcal{U}_\mathcal{F}(\mathcal{P}, \mathcal{R})$ where x_{p+1}, \dots, x_n are fresh variables. Since $\mathcal{U}(\mathcal{P}, \mathcal{R})$ is defined, $\perp \notin \mathcal{U}_\mathcal{F}(\mathcal{P}, \mathcal{R})$, and therefore $\perp \notin \mathcal{U}_\mathcal{F}(r x_{p+1} \dots x_n)[\varphi]$. If $p = n$, due to Lemma 40, $\mathcal{U}_\mathcal{F}^\alpha(t') = \mathcal{U}_\mathcal{F}^\alpha(r\sigma) \subseteq \mathcal{U}_\mathcal{F}(r)[\varphi] \subseteq \mathcal{U}_\mathcal{F}(\mathcal{P}, \mathcal{R})$. Otherwise, $r = g r_1 \dots r_q$ for $g \in \mathcal{F}$. Then $\mathcal{U}_\mathcal{F}^\alpha(t') = \mathcal{U}_\mathcal{F}^\alpha((r\sigma) t_{p+1} \dots t_n) \subseteq \{(g, q+n-p)\} \cup \bigcup \{\mathcal{U}_\mathcal{F}^\alpha(r_i\sigma) \mid 1 \leq i \leq q, i \in \text{regard}(g)\} \cup \bigcup \{\mathcal{U}_\mathcal{F}^\alpha(t_{p+j}) \mid 1 \leq j \leq n-p, q+j \in \text{regard}(g)\}$ and $\mathcal{U}_\mathcal{F}(r x_{p+1} \dots x_n)[\varphi] = \{(g, q+$

$n - p) \} \cup \bigcup \{ \mathcal{U}_{\mathcal{F}}(r_i)[\varphi] \mid 1 \leq i \leq q, i \in \text{regard}(\mathbf{g}) \}$. Since $\perp \notin \mathcal{U}_{\mathcal{F}}(r_{x_{p+1} \cdots x_n})[\varphi]$, for each such i , $\mathcal{U}_{\mathcal{F}}(r_i\sigma) \subseteq \mathcal{U}_{\mathcal{F}}(r_i)[\varphi]$ due to Lemma 40. Since $\mathcal{U}(\mathcal{P}, \mathcal{R})$ is defined, each $p+j \in \text{regard}(\mathbf{f})$, and therefore $\mathcal{U}_{\mathcal{F}}(t_{p+j}) \subseteq \mathcal{U}_{\mathcal{F}}(t)$. Hence, $\mathcal{U}_{\mathcal{F}}(t') \subseteq \mathcal{U}_{\mathcal{F}}(r_{x_{p+1} \cdots x_n})[\varphi] \cup \mathcal{U}_{\mathcal{F}}(t) \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$.

(2) $t = \mathbf{f} \, v_1 \cdots v_n$ where \mathbf{f} is a calculation symbol and v_1, \dots, v_n are theory values, and t has a base type. Then $t \rightarrow_{\kappa} t'$ and t' is a theory value. Hence, $\mathcal{U}_{\mathcal{F}}(t') = \emptyset$.

(3) $t = \mathbf{f} \, t_1 \cdots t_n$ for $\mathbf{f} \in \mathcal{F}^{\sharp}$, and $t' = \mathbf{f} \, t_1 \cdots t_{k-1} \, t'_k \, t_{k+1} \cdots t_n$ for some k with $t_k \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t'_k$. If t is a ground theory term, so is t' , hence $\mathcal{U}_{\mathcal{F}}(t') = \emptyset$; so assume otherwise. If $k \notin \text{regard}(\mathbf{f})$, $\mathcal{U}_{\mathcal{F}}(t') \subseteq \mathcal{U}_{\mathcal{F}}(t) \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$. If $k \in \text{regard}(\mathbf{f})$, $\mathcal{U}_{\mathcal{F}}(t_k) \subseteq \mathcal{U}_{\mathcal{F}}(t) \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$. By induction, $\mathcal{U}_{\mathcal{F}}(t'_k) \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$. Then $\mathcal{U}_{\mathcal{F}}(t') \subseteq \mathcal{U}_{\mathcal{F}}(t) \cup \mathcal{U}_{\mathcal{F}}(t'_k) \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$.

(4) $t = x \, t_1 \cdots t_n$ for $x \in \mathcal{V}$, and $t' = x \, t_1 \cdots t_{k-1} \, t'_k \, t_{k+1} \cdots t_n$; then $\mathcal{U}_{\mathcal{F}}(t') = \emptyset$. \blacktriangleleft

► **Theorem 42.** *Usable-rules processors are sound.*

Proof of Theorem 42. Given a DP problem $(\mathcal{P}, \mathcal{R})$ such that $\mathcal{NF}(\mathcal{Q}) \subseteq \mathcal{NF}(\mathcal{R})$ and $\mathcal{U}(\mathcal{P}, \mathcal{R})$ is defined, and an infinite innermost $(\mathcal{P}, \mathcal{R})$ -chain $(s_0^{\sharp} \Rightarrow t_0^{\sharp} [\varphi_0], \sigma_0), (s_1^{\sharp} \Rightarrow t_1^{\sharp} [\varphi_1], \sigma_1), \dots$ for all i , since $\mathcal{U}_{\mathcal{F}}(t_i^{\sharp})[\varphi_i] \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$, $\perp \notin \mathcal{U}_{\mathcal{F}}(t_i^{\sharp})[\varphi_i]$, and due to Lemma 40, $\mathcal{U}_{\mathcal{F}}(t_i^{\sharp}\sigma_i) \subseteq \mathcal{U}_{\mathcal{F}}(t_i^{\sharp})[\varphi_i] \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$. Now due to Lemma 41, for all t' such that $t_i^{\sharp}\sigma_i \xrightarrow{\mathcal{Q}}_{\mathcal{R}}^* t'$, $\mathcal{U}_{\mathcal{F}}(t') \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$, so any $\xrightarrow{\mathcal{Q}}_{\mathcal{R}}$ -step from t' is a $\xrightarrow{\mathcal{Q}}_{\mathcal{U}(\mathcal{P}, \mathcal{R})}$ -step, given the definition of $\mathcal{U}_{\mathcal{F}}()$. \blacktriangleleft

Below we let π denote $\pi_{\mathbf{t}}$ —the argument filtering with respect to the Boolean \mathbf{t} —and define σ^{π} for each substitution σ by letting $\sigma^{\pi}(x)$ be $\pi(\sigma(x))$.

► **Lemma 43.** *Given a logical constraint φ and a substitution σ such that $\sigma(x)$ is a theory value for all $x \in \text{Var}(\varphi)$, $\pi_{\varphi}(t)\sigma^{\pi} = \pi(t\sigma)$ for each pattern t such that all the variables in $\text{Var}(t)$ whose type is a theory sort are also in $\text{Var}(\varphi)$.*

Proof of Lemma 43. By induction on t . If $t = x \, t_1 \cdots t_n$ with $x \in \mathcal{V}$, then $n = 0$ because t is a pattern. Hence, $\pi_{\varphi}(t)\sigma^{\pi} = \sigma^{\pi}(x) = \pi(\sigma(x)) = \pi(t\sigma)$. Otherwise, $t = \mathbf{f} \, t_1 \cdots t_n$ for $\mathbf{f} \in \mathcal{F}^{\sharp}$. If t is in $T(\mathcal{F}_{\emptyset}, \text{Var}(\varphi))$ and has a base type, $\pi_{\varphi}(t)\sigma^{\pi} = t\sigma^{\pi} = t\sigma = \pi(t\sigma)$ because $\sigma(x)$ is a theory value for all $x \in \text{Var}(t)$ and $\pi(v) = v$ for each theory value v . Otherwise, if $\text{regard}(\mathbf{f}) \cap \{1, \dots, n\} = \{i_1, \dots, i_k\}$ (with $i_1 < \dots < i_k$), $\pi_{\varphi}(t)\sigma^{\pi} = \mathbf{f}_n (\pi_{\varphi}(t_{i_1})\sigma^{\pi}) \cdots (\pi_{\varphi}(t_{i_k})\sigma^{\pi})$ and $\pi(t\sigma) = \mathbf{f}_n \pi(t_{i_1}\sigma) \cdots \pi(t_{i_k}\sigma)$. By induction, $\pi_{\varphi}(t_{i_j})\sigma^{\pi} = \pi(t_{i_j}\sigma)$ for all $j \in \{1, \dots, k\}$. \blacktriangleleft

► **Lemma 44.** *Given a set \mathcal{P} of SDPs, a set \mathcal{R} of rewrite rules and a constrained relation \succeq such that $\mathcal{U}(\mathcal{P}, \mathcal{R})$ is defined, $\ell \succeq_{\varphi} r$ for all $\ell \rightarrow r$ $[\varphi] \in \mathcal{U}(\mathcal{P}, \mathcal{R})$ and \succeq is covered by some reflexive, transitive and monotonic relation \sqsupseteq : if $\mathcal{U}_{\mathcal{F}}(t)[\varphi] \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$ and $\sigma(x)$ is a theory value for all $x \in \text{Var}(\varphi)$, $\pi_{\varphi}(t)\sigma^{\pi} \downarrow_{\kappa} \sqsupseteq \pi(t\sigma) \downarrow_{\kappa}$.*

Proof of Lemma 44. By induction on t . If $t = x \, t_1 \cdots t_n$ with $x \in \mathcal{V}$ then $n = 0$ because $\perp \notin \mathcal{U}_{\mathcal{F}}(t)[\varphi]$. Hence, $\pi_{\varphi}(t)\sigma^{\pi} = \sigma^{\pi}(x) = \pi(\sigma(x)) = \pi(t\sigma)$. Otherwise, $t = \mathbf{f} \, t_1 \cdots t_n$ for $\mathbf{f} \in \mathcal{F}^{\sharp}$. If t is in $T(\mathcal{F}_{\emptyset}, \text{Var}(\varphi))$ and has a base type, $\pi_{\varphi}(t)\sigma^{\pi} = t\sigma^{\pi} = t\sigma = \pi(t\sigma)$ because each $\sigma(x)$ is a theory value so $\pi(\sigma(x)) = \sigma(x)$. Otherwise, we have $\pi_{\varphi}(t)\sigma^{\pi} = \mathbf{f}_n (\pi_{\varphi}(t_{i_1})\sigma^{\pi}) \cdots (\pi_{\varphi}(t_{i_k})\sigma^{\pi})$ where $\text{regard}(\mathbf{f}) \cap \{1, \dots, n\} = \{i_1, \dots, i_k\}$ and $i_1 < \dots < i_k$. By induction, $\pi_{\varphi}(t_{i_j})\sigma^{\pi} \downarrow_{\kappa} \sqsupseteq \pi(t_{i_j}\sigma) \downarrow_{\kappa}$ for all $j \in \{1, \dots, k\}$, and therefore $\pi_{\varphi}(t)\sigma^{\pi} \downarrow_{\kappa} \sqsupseteq \mathbf{f}_n \pi(t_{i_1}\sigma) \downarrow_{\kappa} \cdots \pi(t_{i_k}\sigma) \downarrow_{\kappa}$. If $t\sigma$ is a ground theory term with a base type, $\mathbf{f}_n x_{i_1} \cdots x_{i_k} \rightarrow y$ $[y = \mathbf{f} \, x_1 \cdots x_n] \in \mathcal{U}(\mathcal{P}, \mathcal{R})$; because \sqsupseteq covers \succeq , $\mathbf{f}_n \pi(t_{i_1}\sigma) \downarrow_{\kappa} \cdots \pi(t_{i_k}\sigma) \downarrow_{\kappa} = \mathbf{f}_n (t_{i_1}\sigma) \downarrow_{\kappa} \cdots (t_{i_k}\sigma) \downarrow_{\kappa} \sqsupseteq (t\sigma) \downarrow_{\kappa} = \pi(t\sigma) \downarrow_{\kappa}$. Otherwise, we have $\pi(t\sigma) = \mathbf{f}_n \pi(t_{i_1}\sigma) \cdots \pi(t_{i_k}\sigma)$. \blacktriangleleft

► **Lemma 45.** *Given a set \mathcal{P} of SDPs, a set \mathcal{R} of rules and a constrained relation \succeq where (1) $\mathcal{NF}(\mathcal{Q}) \subseteq \mathcal{NF}(\mathcal{R})$, (2) $\mathcal{U}(\mathcal{P}, \mathcal{R})$ is defined, (3) for all $\ell \rightarrow r$ $[\varphi] \in \mathcal{R}$, all $x \in \text{Var}(\ell)$ whose type is a theory sort, $x \in \text{Var}(\varphi)$, (4) $\ell \succeq_{\varphi} r$ for all $\ell \rightarrow r$ $[\varphi] \in \mathcal{U}(\mathcal{P}, \mathcal{R})$ and (5) \succeq is covered by a reflexive, transitive, monotonic \sqsupseteq : if $\mathcal{U}_{\mathcal{F}}^{\alpha}(t) \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$ and $t \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t'$, $\pi(t) \downarrow_{\kappa} \sqsupseteq \pi(t') \downarrow_{\kappa}$.*

Proof of Lemma 45. By induction on t .

- (1) t is a base-type ground theory term; then so is t' , and $t \rightarrow_{\kappa} t'$. Hence, $\pi(t) \downarrow_{\kappa} = \pi(t') \downarrow_{\kappa}$.
- (2) $t = f \ t_1 \cdots t_n$ for $f \in \mathcal{F}$, and there exist a number $p \leq n$, substitution σ and rule $\ell \rightarrow r \ [\varphi] \in \mathcal{R}$ such that $\ell = f \ \ell_1 \cdots \ell_p$, $t_i = \ell_i \sigma$ for all $i \in \{1, \dots, p\}$, $t' = (r\sigma) \ t_{p+1} \cdots t_n$, $\sigma(x)$ is in normal form for all x and σ respects φ . Since $\mathcal{NF}(\mathcal{Q}) \subseteq \mathcal{NF}(\mathcal{R})$, $(f, n) \in \mathcal{U}_{\mathcal{F}}^a(t) \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$, and therefore $\mathcal{U}_{\mathcal{F}}(r \ x_{p+1} \cdots x_n)[\varphi] \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$ and $\pi_{\varphi}(\ell \ x_{p+1} \cdots x_n) \rightarrow \pi_{\varphi}(r \ x_{p+1} \cdots x_n) \ [\varphi] \in \mathcal{U}(\mathcal{P}, \mathcal{R})$ for x_{p+1}, \dots, x_n fresh variables. We have $\pi(t) = f_n \ \pi(t_{i_1}) \cdots \pi(t_{i_k})$ where $\text{regard}(f) \cap \{1, \dots, n\} = \{i_1, \dots, i_k\}$ and $i_1 < \dots < i_l \leq p < i_{l+1} < \dots < i_k$. By Lemma 43, $\pi_{\varphi}(\ell_j) \sigma^{\pi} = \pi(\ell_j \sigma) = \pi(t_j)$ for all $j \in \{1, \dots, l\}$. Since $\pi_{\varphi}(x_j)([x_i :- t_i]_{i=p+1}^n)^{\pi} = \pi(t_j)$ for all $j \in \{l+1, \dots, k\}$, $\pi(t) = \pi_{\varphi}(\ell \ x_{p+1} \cdots x_n)(\sigma \cup [x_i :- t_i]_{i=p+1}^n)^{\pi}$. Due to Lemma 44, $\pi_{\varphi}(r \ x_{p+1} \cdots x_n)(\sigma \cup [x_i :- t_i]_{i=p+1}^n)^{\pi} \downarrow_{\kappa} \sqsupseteq \pi(t') \downarrow_{\kappa}$. Because \sqsupseteq covers \succeq and $(\sigma \cup [x_i :- t_i]_{i=p+1}^n)^{\pi}$ respects φ , $\pi(t) \downarrow_{\kappa} \sqsupseteq \pi(t') \downarrow_{\kappa}$.
- (3) $t = f \ t_1 \cdots t_n$ ($f \in \mathcal{F}^{\#}$) is not a base-type ground theory term, and there is m such that $t' = f \ t_1 \cdots t_{m-1} \ t'_m \ t_{m+1} \cdots t_n$ and $t_m \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t'_m$. Then $\pi(t) = f_n \ \pi(t_{i_1}) \cdots \pi(t_{i_k})$ where $\text{regard}(f) \cap \{1, \dots, n\} = \{i_1, \dots, i_k\}$ and $i_1 < \dots < i_k$. Let t'_i be t_i for all $i \in \{1, \dots, n\} \setminus \{m\}$. By induction, $\pi(t_m) \downarrow_{\kappa} \sqsupseteq \pi(t'_m) \downarrow_{\kappa}$, and therefore $\pi(t) \downarrow_{\kappa} \sqsupseteq f_n \ \pi(t'_{i_1}) \downarrow_{\kappa} \cdots \pi(t'_{i_k}) \downarrow_{\kappa}$. If t' is a ground theory term that has a base type, since $(f, n) \in \mathcal{U}_{\mathcal{F}}^a(t) \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$, we have $f_n \ x_{i_1} \cdots x_{i_k} \rightarrow y \ [y = f \ x_1 \cdots x_n] \in \mathcal{U}(\mathcal{P}, \mathcal{R})$; because \sqsupseteq covers \succeq , $f_n \ \pi(t'_{i_1}) \downarrow_{\kappa} \cdots \pi(t'_{i_k}) \downarrow_{\kappa} = f_n \ t'_{i_1} \downarrow_{\kappa} \cdots t'_{i_k} \downarrow_{\kappa} \sqsupseteq t' \downarrow_{\kappa} = \pi(t') \downarrow_{\kappa}$. Otherwise, $\pi(t') \downarrow_{\kappa} = f_n \ \pi(t'_{i_1}) \downarrow_{\kappa} \cdots \pi(t'_{i_k}) \downarrow_{\kappa}$.
- (4) $t = x \ t_1 \cdots t_k \cdots t_n$ for $x \in \mathcal{V}$, and $t' = x \ t_1 \cdots t'_k \cdots t_n$; then $\pi(t) = \bullet_{\mu(A)} = \pi(t')$. \blacktriangleleft

► **Theorem 46.** *If the rewrite rules are preprocessed by Lemma 7, and all theory sorts are inextensible, then reduction pair processors with an argument filtering are sound.*

Proof of Theorem 46. Following Theorem 42, for all i and t' such that $t_i^{\#} \sigma_i \xrightarrow{\mathcal{Q}}_{\mathcal{R}}^* t'$, $\mathcal{U}_{\mathcal{F}}^a(t') \subseteq \mathcal{U}_{\mathcal{F}}(\mathcal{P}, \mathcal{R})$. Due to Lemma 45, $\pi(t_i^{\#} \sigma_i) \downarrow_{\kappa} \sqsupseteq \pi(s_{i+1}^{\#} \sigma_{i+1}) \downarrow_{\kappa}$ for all i . For all i , if $s_i^{\#} \Rightarrow t_i^{\#} [\varphi_i] \in \mathcal{P}'$, due to Lemmas 43 and 44, $\pi(s_i^{\#} \sigma_i) \downarrow_{\kappa} = \pi_{\varphi_i}(s_i^{\#} \sigma_i)^{\pi} \downarrow_{\kappa} \sqsupseteq \pi_{\varphi_i}(t_i^{\#} \sigma_i)^{\pi} \downarrow_{\kappa} \sqsupseteq \pi(t_i^{\#} \sigma_i) \downarrow_{\kappa}$; if $s_i^{\#} \Rightarrow t_i^{\#} [\varphi_i] \in \mathcal{P} \setminus \mathcal{P}'$, similarly $\pi(s_i^{\#} \sigma_i) \downarrow_{\kappa} \sqsupseteq \pi(t_i^{\#} \sigma_i) \downarrow_{\kappa}$. Since \sqsupseteq is well-founded, an infinite innermost $(\mathcal{P}, \mathcal{R})$ -chain leads to an infinite innermost $(\mathcal{P} \setminus \mathcal{P}', \mathcal{R})$ -chain. \blacktriangleleft

A.2 The Proofs of Theorems 14 and 36

For the properties of \mathbb{C} -computability, see Appendix A in the extended version of [13]. The following proofs are largely based on Appendix A.3 in [20]; here the novelty lies on the innermostness of chains. Note that undefined function symbols are \mathbb{C} -computable (see [20]).

► **Lemma 47.** *Assume 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 $f : A_1 \rightarrow \cdots \rightarrow A_m \rightarrow B$ 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 , then there exist an SDP $f^{\#} \ s'_1 \cdots s'_m \Rightarrow g^{\#} \ t_1 \cdots t_n \ [\varphi] \in \text{SDP}(\mathcal{R}_0)$, a substitution σ and a natural number p such that (1) $s_i \xrightarrow{i}_{\mathcal{R}_0 \cup \mathcal{R}_1} s'_i \sigma$ and $s'_i \sigma$ is in normal form for all $i \leq p$, (2) s'_i is a fresh variable (s'_i does not occur in s'_j for any $j \neq i$) and $\sigma(s'_i) = s_i$ for all $i > p$, (3) σ respects φ , and (4) $(g \ t_1 \cdots t_n) \sigma = g \ (t_1 \sigma) \cdots (t_n \sigma)$ is not \mathbb{C} -computable in $\mathcal{R}_0 \cup \mathcal{R}_1$ with \succsim' but $u \sigma$ is for each proper subterm u of $g \ t_1 \cdots t_n$.*

Proof. If the only reducts of $f \ s_1 \cdots s_m$ were values or $f \ s'_1 \cdots s'_m$ with $s_i \xrightarrow{i}_{\mathcal{R}_0 \cup \mathcal{R}_1} s'_i$ for all i then $f \ s_1 \cdots s_m$ would be computable. Hence, there exist $f \ s'_1 \cdots s'_p \rightarrow r \ [\varphi] \in \mathcal{R}_0$ (f cannot be defined in \mathcal{R}_1) and σ' such that $s_i \xrightarrow{i}_{\mathcal{R}_0 \cup \mathcal{R}_1} s'_i \sigma'$ and $s'_i \sigma'$ is in normal form for all $i \leq p$, and σ' respects φ ; $(r \sigma') \ s_{p+1} \cdots s_m$ is thus a reduct of $f \ s_1 \cdots s_m$. At least one such reduct is uncomputable. Let $(r \sigma') \ s_{p+1} \cdots s_m$ be uncomputable, and therefore so is $r \sigma'$. Also all $\sigma'(x)$ are computable, since these are accessible subterms of some s'_i , since \mathcal{R}_0 is AFP.

Take a minimal subterm $a \ t_1 \cdots t_q$ of r such that $t\sigma'$ is uncomputable. By minimality, each $t_i\sigma'$ is computable. Hence, a cannot be a variable, value or constructor, as then $\sigma'(x)$ would be computable, implying computability of $t\sigma'$. Hence, a is a defined symbol g .

We have $f^\# \ s'_1 \cdots s'_p \ x_{p+1} \cdots x_m \Rightarrow g^\# \ t_1 \cdots t_q \ y_{q+1} \cdots y_n \ [\varphi] \in \text{SDP}(\mathcal{R}_0)$. Because $t\sigma'$ is uncomputable, there exist computable terms t'_{q+1}, \dots, 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 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 z . Let s'_i denote x_i for $i > p$ and t_i denote y_i for $i > q$; then $f^\# \ s'_1 \cdots s'_m \Rightarrow g^\# \ t_1 \cdots t_n \ [\varphi]$, σ and p satisfy all requirements. \blacktriangleleft

► **Corollary 48.** *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 $f : A_1 \rightarrow \cdots \rightarrow A_m \rightarrow B$ in \mathcal{R}_0 (B 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 each s_i is, there exists an infinite innermost $(\text{SDP}(\mathcal{R}_0), \mathcal{R}_0 \cup \mathcal{R}_1)$ -chain $(f^\# \ s'_1 \cdots s'_m \Rightarrow t^\# \ [\varphi], \sigma), \dots$*

Proof. Repeatedly applying Lemma 47 (on $f \ s_1 \cdots s_m$, then on $g \ (t_1\sigma) \cdots (t_n\sigma)$ and so on) we get three infinite sequences: one of SDPs in $\text{SDP}(\mathcal{R}_0)$ $(f_i^\# \ s_{i,1} \cdots s_{i,m_i} \Rightarrow f_{i+1}^\# \ t_{i,1} \cdots t_{i,n_i} \ [\varphi_i])_i$ where $f_0 = f$, one of substitutions $(\sigma_i)_i$ and one of natural numbers $(p_i)_i$. Since $\xrightarrow{i} \mathcal{R}_0 \cup \mathcal{R}_1$ is exactly $\xrightarrow{\mathcal{R}_0 \cup \mathcal{R}_1} \mathcal{R}_0 \cup \mathcal{R}_1$, combining the SDPs and the substitutions almost gives us an infinite innermost $(\text{SDP}(\mathcal{R}_0), \mathcal{R}_0 \cup \mathcal{R}_1)$ -chain, except that $s_{i,j}\sigma_i$ may not be in normal form if $j > p_i$.

We define $(\sigma'_i)_i$ so that, together with the SDPs, we obtain an infinite chain. For all i, j with $p_i < j \leq m_i$, $s_{i,j}$ is a fresh variable and $\sigma_i(s_{i,j})$ is \mathbb{C} -computable. Let $\sigma'_i(x)$ be $\sigma_i(x)$ for $x \in \mathcal{V} \setminus \{s_{i,p_i+1}, \dots, s_{i,m_i}\}$. To define $\sigma'_i(s_{i,j})$ for $j > p_i$, let k_0 be j and for $l \geq 0$, if k_l exists and $k_l > p_{i+l}$, let k_{l+1} be the index (if any) with $s_{i+l,k_l} = t_{i+l,k_{l+1}}$. There are two options:

(1) If $k_l > p_{i+l}$ for all l such that k_l is defined, regardless of whether the sequence is finite, $\sigma_{i+l}(s_{i+l,k_l}) = \sigma_{i+l}(t_{i+l,k_{l+1}}) = \sigma_{i+l+1}(s_{i+l+1,k_{l+1}})$ for all l such that k_{l+1} is defined. Let u be an arbitrary normal form of $\sigma_i(s_{i,j})$, and let $\sigma'_{i+l}(s_{i+l,k_l})$ be u for all l where k_l is defined.

(2) Otherwise, the sequence $(k_l)_l$ is finite, and $k_q \leq p_{i+q}$ where k_q is the last in the sequence. Then $s_{i+q,k_q}\sigma_{i+q}$ is in normal form. Let $\sigma'_{i+l}(s_{i+l,k_l})$ be $s_{i+q,k_q}\sigma_{i+q}$ for all $l < q$.

Hence, we have built an infinite innermost $(\text{SDP}(\mathcal{R}_0), \mathcal{R}_0 \cup \mathcal{R}_1)$ -chain. \blacktriangleleft

► **Theorem 14.** *An AFP system \mathcal{R} is innermost (and therefore call-by-value) terminating if there exists no infinite innermost $(\text{SDP}(\mathcal{R}), \mathcal{R})$ -chain.*

Proof. Towards a contradiction, assume that \mathcal{R} is not innermost terminating. Then there is an uncomputable (not \mathbb{C} -computable) term u . Take a minimal subterm s of u that is uncomputable; then $u = f \ s_1 \cdots s_k$ where f is defined and each s_i is computable. Let $f : A_1 \rightarrow \cdots \rightarrow A_m \rightarrow B$ for $B \in \mathcal{S}$. Since $s = f \ s_1 \cdots s_k$ is uncomputable, there exist computable s_{k+1}, \dots, s_m with $s \ s_{k+1} \cdots s_m = f \ s_1 \cdots s_m$ uncomputable. By Corollary 48 (with $\mathcal{R}_1 = \emptyset$), there is an infinite innermost $(\text{SDP}(\mathcal{R}), \mathcal{R})$ -chain, giving the required contradiction. \blacktriangleleft

► **Theorem 36.** *An accessible function passing system \mathcal{R}_0 with sort ordering \succsim is universally computable if there exists no infinite innermost $(\text{SDP}(\mathcal{R}_0), \mathcal{R}_0 \cup \mathcal{R}_1)$ -chain for any extension \mathcal{R}_1 of \mathcal{R}_0 and extension \succsim' of \succsim to sorts in $\mathcal{R}_0 \cup \mathcal{R}_1$.*

Proof. Towards a contradiction, assume that \mathcal{R}_0 is not universally computable. There exist an extension \mathcal{R}_1 of \mathcal{R}_0 , sort ordering \succsim' which extends \succsim over sorts in $\mathcal{R}_0 \cup \mathcal{R}_1$ and term u of \mathcal{R}_0 that is not \mathbb{C} -computable in $\mathcal{R}_0 \cup \mathcal{R}_1$ with \succsim' . We consider \mathbb{C} -computability in $\mathcal{R}_0 \cup \mathcal{R}_1$. Take a minimal uncomputable subterm s of u ; then s must take the form $f \ s_1 \cdots s_k$ with f a defined symbol in \mathcal{R}_0 and each s_i computable. Let $f : A_1 \rightarrow \cdots \rightarrow A_m \rightarrow B$. Because s is uncomputable, there exist computable s_{k+1}, \dots, s_m of $\mathcal{R}_0 \cup \mathcal{R}_1$ with $s \ s_{k+1} \cdots s_m = f \ s_1 \cdots s_m$ uncomputable. Corollary 48 gives the required chain to obtain a contradiction. \blacktriangleleft