Cutting a proof into bite-sized chunks

Incrementally proving termination in higher-order term rewriting

Cynthia Kop
Department of Software Science, Radboud University Nijmegen, The Netherlands

Abstract

This paper discusses a number of methods to prove termination of higher-order term rewriting systems, with a particular focus on *large* systems. In first-order term rewriting, the dependency pair framework can be used to split up a large termination problem into multiple (much) smaller components that can be solved individually. This is important because a large problem may take exponentially longer to solve in one go than solving each of its components.

Unfortunately, while there are higher-order versions of several of these methods, they often fail to simplify a problem enough. Here, we will explore some of these techniques and their limitations, and discuss what else can be done to incrementally build a termination proof for higher-order systems.

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

Keywords and phrases Termination, Modularity, Higher-order term rewriting, Dependency Pairs, Algebra Interpretations

Digital Object Identifier 10.4230/LIPIcs.FSCD.2022.32

Category Invited Talk

Funding The author is supported by the NWO TOP project "ICHOR", NWO 612.001.803/7571 and the NWO VIDI project "CHORPE", NWO VI.Vidi.193.075.

1 Introduction

In the last few decades, the term rewriting community has developed a wide scala of techniques to prove termination of term rewriting systems. A variety of automatic termination analysis tools compete against each other in the annual termination competition [23], using hundreds of different techniques. Many of these techniques can be adapted to other forms of rewriting (e.g., context-sensitive, conditional), or real-world programming languages.

Higher-order term rewriting systems in particular are very close to functional programming languages, and ideas developed in one are likely to extend to the other. However, realistic (functional) programs often have thousands of lines. Many termination techniques are ill-equipped for this. For example, naively finding a suitable polynomial interpretation or path ordering is exponential in the size of the TRS.

Ideally, we would like to split up a large TRS into many small parts; prove termination of each, and conclude termination of the whole. Unfortunately, this is in general impossible, as termination is not modular [21]. Instead, we may look to different properties than termination. The *dependency pair framework* [12] is a de facto standard for termination proofs in first-order term rewriting, which combines various techniques to do exactly this: a termination problem is translated into one or more *DP problems*, which are gradually simplified, split up, and eventually closed, without ever having to apply an exponential technique on all rules at once.

The DP framework has been extended to higher-order rewriting [1, 11, 16, 18]. However, some methods in the framework adapt poorly to higher-order rules; in particular *usable rules* – an important technique to remove large numbers of rules from a DP problem – are likely to fail. Hence, even with dependency pairs, we often need to find an ordering for thousands of rules at once. Hence, it seems important to develop incremental ways to find an ordering.

In this paper, I will highlight how higher-order dependency pairs can be used to cut termination proofs into (potentially many) smaller proof obligations, and where this approach is weak. In addition, I will sketch a way to incrementally build a term ordering using *tuple interpretations* [17], a recently developed methodology based on algebra interpretations [10, 20] which was designed for *complexity analysis*, but also proves very powerful for termination.

Contribution. This paper introduces usable rules with respect to an argument filtering for higher-order term rewriting, and lifts the arity restrictions in weakly monotonic interpretations [10]. However, the purpose of this paper is not to introduce new theory, but rather to explain how known techniques can be applied to build up a higher-order termination proof in many small steps. Hence, we will focus on a simple format that allows for an easy presentation.

Related work. Aside from various definitions of dependency pairs, the most relevant related work is a recent approach by Hamana [13] which aims to split up a TRS into two parts: one which should be proved terminating when combined with some simple additional rules, the other ordered by a specific technique. This is discussed a bit further in Section 4.

2 Preliminaries

Unlike first-order term rewriting, there is no single, unified approach to higher-order term rewriting, but rather a number of similar but not fully compatible systems aiming to combine term rewriting and typed λ-calculi. Since this paper aims to explain *ideas* rather than provide technical detail, we will use a formalism that allows for a simple presentation: simply-typed λ-calculus with base-type rules and plain matching. The ideas extend to other forms of higher-order rewriting, but most definitions (e.g., dependency pairs) need more cases there.

Given a set $\mathbb{S}$ of *sorts*, the set $\mathbb{ST}$ of *simple types* is given by: (a) $\mathbb{S} \subseteq \mathbb{ST}$ and (b) if $\sigma, \tau \in \mathbb{ST}$ then $\sigma \Rightarrow \tau \in \mathbb{ST}$. Types are denoted σ, τ, ρ and sorts ι, κ. We let $\Rightarrow$ be right-associative. Hence, all types have a unique representation in the form $\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota$.

We assume given disjoint sets $\mathcal{F}$ of typed function symbols, notation $(\mathtt{f} :: \sigma) \in \mathcal{F}$, and $\mathcal{V}$ of typed variables, notation $(x :: \sigma) \in \mathcal{V}$; there should be countably many variables of each type. *Terms* are expressions s where $s :: \sigma$ can be inductively derived for some σ by: (a) $a :: \sigma$ if $(a :: \sigma) \in \mathcal{F} \cup \mathcal{V}$; (b) $s\ t :: \tau$ if $s :: \sigma \Rightarrow \tau$ and $t :: \sigma$; (c) $\lambda x.s :: \sigma \Rightarrow \tau$ if $(x :: \sigma) \in \mathcal{V}$ and $s :: \tau$. The λ binds variables as in the λ-calculus; unbound variables are called *free* and $\mathcal{FV}(s)$ is the set of variables occurring unbound in s. A term s is called *closed* if $\mathcal{FV}(s) = \emptyset$. Term equality is modulo α-conversion. Application is left-associative. A term s *has type* σ if $s :: \sigma$; it *has base type* if $\sigma \in \mathbb{S}$. The *head symbol* of a term $\mathtt{f}\ s_1 \cdots s_n$ is $\mathtt{f}$.

A term s has a *maximally applied subterm* t, notation $s \trianglerighteq t$, if either $s = t$, or $s \triangleright t$, where $s \triangleright t$ if (a) $s = a\ s_1 \cdots s_n$ with $a \in \mathcal{F} \cup \mathcal{V}$ and some $s_i \trianglerighteq t$; or (b) $s = (\lambda x.u)\ s_1 \cdots s_n$ (with $n \geq 0$) and some $s_i \trianglerighteq t$ or $u \trianglerighteq t$. Note that *not* $s\ t \trianglerighteq s$. A *pattern* is a term s such that whenever $s \trianglerighteq t\ s_1 \cdots s_n$ with $n > 0$ then t is not an abstraction or an element of $\mathcal{FV}(s)$.

A substitution is a type-preserving mapping from variables to terms. The *domain* of a substitution γ is the set $\{x \in \mathcal{V} \mid \gamma(x) \neq x\}$. Substitution does not capture bound variables; we let: (a) $x\gamma = \gamma(x)$; (b) $\mathtt{f}\gamma = \mathtt{f}$; (c) $(s\ t)\gamma = (s\gamma)\ (t\gamma)$ and (d) $(\lambda x.s)\gamma = \lambda x.(s\gamma)$ if $\gamma(x) = x$ and there is no y such that $x \in \mathcal{FV}(\gamma(y))$; this is always defined by α-conversion.

A relation $\to$ on terms is *monotonic* if $s \to t$ implies $\lambda x.s \to \lambda x.t$ and $u\ s \to u\ t$ and $s\ u \to t\ u$. The relation $\to_\beta$ is the smallest monotonic relation such that $(\lambda x.s)\ t \to_\beta s[x := t]$, where $[x := t]$ is the substitution mapping x to t. A *rewrite rule* is a pair $\ell \to r$ of a *pattern* ℓ of the form $\mathtt{f}\ \ell_1 \cdots \ell_k$ and a term r such that $\mathcal{FV}(r) \subseteq \mathcal{FV}(\ell)$, ℓ and r have the same **base type**, and r has no subterms of the form $(\lambda x.s)\ t_1 \cdots t_n$ with $n > 0$. Given a set of rules

$\mathcal{R}$, the relation $\to_{\mathcal{R}}$ is the smallest monotonic relation on terms such that $\ell\gamma \to_{\mathcal{R}} r\gamma$ for all $\ell \to r \in \mathcal{R}$ and substitutions γ, and $\to_{\mathcal{R}}$ includes $\to_\beta$. A term s is *in normal form* if there is no t such that $s \to_{\mathcal{R}} t$, and it is *β-normal* if there is no t such that $s \to_\beta t$. It is *terminating* if there is no infinite reduction $s \to_{\mathcal{R}} s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} \dots$. We say that $\to_{\mathcal{R}}$ is terminating if all terms over $\mathcal{F}, \mathcal{V}$ are terminating. The set $\mathcal{D} \subseteq \mathcal{F}$ of *defined symbols* consists of those $\mathtt{f}$ such that $\mathcal{R}$ contains a rule $\mathtt{f}\ \ell_1 \cdots \ell_k \to r$; all other symbols are called *constructors*.

▶ Remark 1. Note that the limitation that rules have base type is not standard in the higher-order literature. We use it here to support a simpler presentation of definitions.

▶ **Example 2.** As a running example, we will use a system over sorts nat (natural numbers), bool (booleans) and list (lists of numbers). Let $\mathtt{0} :: \mathsf{nat}$, $\mathtt{s} :: \mathsf{nat} \Rightarrow \mathsf{nat}$, $\top :: \mathsf{bool}$, $\bot :: \mathsf{bool}$, $\mathtt{nil} :: \mathsf{list}$, $\mathtt{cons} :: \mathsf{nat} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}$; the types of other symbols can be deduced.

$$\begin{array}{rclrcl}
\mathtt{map}\ F\ \mathtt{nil} & \to & \mathtt{nil} & \mathtt{map}\ F\ (\mathtt{cons}\ x\ a) & \to & \mathtt{cons}\ (F\ x)\ (\mathtt{map}\ F\ a) \\
\mathtt{fold}\ F\ x\ \mathtt{nil} & \to & x & \mathtt{fold}\ F\ x\ (\mathtt{cons}\ y\ a) & \to & \mathtt{fold}\ F\ (F\ x\ y)\ a \\
\mathtt{min}\ x\ \mathtt{0} & \to & x & \mathtt{min}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) & \to & \mathtt{min}\ x\ y \\
\mathtt{quot}\ \mathtt{0}\ (\mathtt{s}\ y) & \to & \mathtt{0} & \mathtt{quot}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) & \to & \mathtt{s}\ (\mathtt{quot}\ (\mathtt{min}\ x\ y)\ (\mathtt{s}\ y)) \\
\mathtt{ack}\ \mathtt{0}\ y & \to & \mathtt{s}\ y & \mathtt{ack}\ (\mathtt{s}\ x)\ \mathtt{0} & \to & \mathtt{ack}\ x\ (\mathtt{s}\ \mathtt{0}) \\
\mathtt{inc}\ \mathtt{0} & \to & \mathtt{s}\ (\mathtt{inc}\ (\mathtt{s}\ \mathtt{0})) & \mathtt{ack}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) & \to & \mathtt{ack}\ x\ (\mathtt{ack}\ (\mathtt{s}\ x)\ y) \\
\mathtt{exp}\ \mathtt{0}\ y & \to & y & \mathtt{exp}\ (\mathtt{s}\ x)\ y & \to & \mathtt{double}\ x\ y\ \mathtt{0} \\
\mathtt{double}\ x\ \mathtt{0}\ z & \to & \mathtt{exp}\ x\ z & \mathtt{double}\ x\ (\mathtt{s}\ y)\ z & \to & \mathtt{double}\ x\ y\ (\mathtt{s}\ (\mathtt{s}\ z)) \\
\mathtt{mkbig}\ a\ x & \to & \mathtt{map}\ (\mathtt{ack}\ x)\ a & \mathtt{mkdiv}\ a\ x & \to & \mathtt{map}\ (\lambda y.\mathtt{quot}\ y\ x)\ a \\
\mathtt{sma}\ b\ F\ \mathtt{0} & \to & \mathtt{0} & \mathtt{sma}\ \top\ F\ (\mathtt{s}\ x) & \to & \mathtt{s}\ x \\
\mathtt{sma}\ \bot\ F\ (\mathtt{s}\ x) & \to & \mathtt{sma}\ (F\ x)\ F\ (\mathtt{quot}\ x\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0}))) & & &
\end{array}$$

In examples in this paper, we let $\mathcal{R}_{\mathtt{f}}$ denote the subset of these rules with only the rules defining $\mathtt{f}$. For example, $\mathcal{R}_{\mathtt{map}}$ refers to the top two rules, and $\mathcal{R}_{\mathtt{ack}}$ has three rules.

Accessibility. Given a quasi-ordering $\succeq^{\mathbb{S}}$ on $\mathbb{S}$ whose strict part $\succ^{\mathbb{S}} := \succeq^{\mathbb{S}} \setminus \preceq^{\mathbb{S}}$ is well-founded, we define, for sort ι and type $\sigma \equiv \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa$, two relations: $\iota \succeq^{\mathbb{S}}_{+} \sigma$ if $\iota \succeq^{\mathbb{S}} \kappa$ and $\iota \succ^{\mathbb{S}}_{-} \sigma_i$ for all i, and $\iota \succ^{\mathbb{S}}_{-} \sigma$ if $\iota \succ^{\mathbb{S}} \kappa$ and $\iota \succeq^{\mathbb{S}}_{+} \sigma_i$ for all i. (Here, $\iota \succeq^{\mathbb{S}}_{+} \sigma$ corresponds to "ι occurs only positively in σ" in [3, 4, 6].) For $\mathtt{f} :: \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \iota$, let $Acc(\mathtt{f}) = \{i \in \{1, \dots, m\} \mid \iota \succeq^{\mathbb{S}} \sigma_i\}$ For terms s, t, denote $s \trianglerighteq_{\mathrm{acc}} t$ if (a) $s = t$, (b) $s = \lambda x.s'$ and $s' \trianglerighteq_{\mathrm{acc}} t$, or (c) $s = \mathtt{f}\ s_1 \cdots s_n$ and $s_i \trianglerighteq_{\mathrm{acc}} t$ for some $i \in Acc(\mathtt{f})$.

For a fixed quasi-ordering $\succeq^{\mathbb{S}}$ on sorts, a term $s :: \iota$ is *computable* iff (1) s is terminating, and (2) if $s \to^*_{\mathcal{R}} \mathtt{f}\ s_1 \cdots s_m$ then s_i is computable for all $i \in Acc(\mathtt{f})$. A term $s :: \sigma \Rightarrow \tau$ is computable iff $s\ t$ is computable for all computable terms $t :: \sigma$. Although this is not an inductive definition, computability is a definable property (see, e.g., [11]).

▶ **Example 3.** For $\mathtt{f} :: (\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{nat}$, we have $Acc(\mathtt{f}) = \emptyset$ for any $\succeq^{\mathbb{S}}$. If $\mathsf{ord} \succ^{\mathbb{S}} \mathsf{nat}$ and $\mathtt{g} :: (\mathsf{nat} \Rightarrow \mathsf{ord}) \Rightarrow \mathsf{ord}$, then we do have $Acc(\mathtt{g}) = \{1\}$. Hence, $\mathtt{f}\ F \not\trianglerighteq_{\mathrm{acc}} F$ but $\mathtt{g}\ F \trianglerighteq_{\mathrm{acc}} F$.

Functions and orderings. A well-founded set is a tuple $(A, >, \geq)$ such that $>$ is a well-founded ordering on A; $\geq$ is a quasi-ordering on A; $x > y$ implies $x \geq y$; and $x > y \geq z$ implies $x > z$. Hence, it is not required that $\geq$ is the reflexive closure of $>$. If $(A_1, >_1 \geq_1)$, $\dots, (A_n, >_n \geq_n)$ are all well-founded sets, then so is $(A_1 \times \cdots \times A_n, >^\times, \geq^\times)$, where $\vec{a} \geq^\times \vec{b}$ if each $a_i \geq_i b_i$, and $\vec{a} >^\times \vec{b}$ if in addition $a_i >_i b_i$ for some i (writing $\vec{a} := \langle a_1, \dots, a_n \rangle$).

Let $(A, >, \geq)$ and $(B, \succ, \succeq)$ be well-founded sets. $A \Longrightarrow B$ is the set of functions from A to B. Function equality is extensional: for $f, g \in A \Longrightarrow B$ we say $f = g$ iff $f(x) = g(x)$ for all $x \in A$. Elements of $A \Longrightarrow B$ are compared pointwise: $f \sqsupset g$ if $f(x) \succ g(x)$ for all $x \in A$; and $f \sqsupseteq g$ if $f(x) \succeq g(x)$ for all $x \in A$. We say that $f \in A \Longrightarrow B$ is *weakly monotonic* if $x \geq y$ implies $f(x) \succeq g(y)$. It is *strongly monotonic* if in addition $x > y$ implies $f(x) \succ g(y)$.

3 Dependency pairs

The traditional way to prove termination of a TRS is to embed the rewrite relation in a well-founded ordering. This is typically done by defining a *monotonic*, *stable* ordering (stable: if $s \succ t$ then $s\gamma \succ t\gamma$ for all substitutions γ), and then showing that $\ell \succ r$ for all rules $\ell \to r$.

▶ **Example 4.** One ordering method is to map each base-type term s to a natural number $[\![s]\!]$, and let $s \succ t$ if $[\![s]\!] > [\![t]\!]$. For example, for some of the symbols in Ex. 2, we may define:

$$\begin{array}{rclrcl} [\![\mathtt{nil}]\!] & = & 0 & [\![\mathtt{map}\ F\ L]\!] & = & ([\![L]\!]+1) * ([\![F]\!]([\![L]\!]) + 1) \\ [\![\mathtt{cons}\ H\ T]\!] & = & [\![H]\!] + [\![T]\!] + 1 & & & \end{array}$$

Here, a term $F :: \mathsf{nat} \Rightarrow \mathsf{nat}$ is mapped to a *strongly monotonic* function in $\mathbb{N} \Rightarrow \mathbb{N}$. We can prove that $[\![\ell]\!] > [\![r]\!]$ holds for the two rules in $\mathcal{R}_{\mathtt{map}}$. Since the interpretation functions are strongly monotonic, and the method is stable by its nature, this shows termination of $\mathcal{R}_{\mathtt{map}}$.

Unfortunately, to prove termination in this way we must find an interpretation that orders all rules at the same time. In a system with thousands of rules, this may well be infeasible. We can do a bit better with *rule removal*: if $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ and we have a (monotonic, stable) well-founded ordering $\succ$ and a compatible (monotonic, stable) quasi-ordering $\succeq$ on terms, and if $\ell \succ r$ for $\ell \to r \in \mathcal{R}_1$ and $\ell \succeq r$ for $\ell \to r \in \mathcal{R}_2$, then $\to_{\mathcal{R}}$ terminates if and only if $\to_{\mathcal{R}_2}$ does. Hence, having a termination proof for $\to_{\mathcal{R}_2}$ makes the termination proof for $\to_{\mathcal{R}}$ easier. However, we still have to orient all rules in $\mathcal{R}$ at once, and $\ell \succeq r$ is often not *that* much easier to show than $\ell \succ r$, partially due to the monotonicity requirement on $\succ$.

▶ **Example 5.** Commonly used orderings like the recursive path ordering and interpretations to $\mathbb{N}$ cannot handle the `quot` rules from Example 2, as the monotonicity requirement on $\succ$ essentially causes the property that, for any choice of ordering/interpretation, $\mathtt{min}\ x\ y \succeq y$; and therefore $\mathtt{quot}\ (\mathtt{s}\ x)\ (\mathtt{s}\ (\mathtt{s}\ x)) \succ \mathtt{s}\ (\mathtt{quot}\ (\mathtt{s}\ x)\ (\mathtt{s}\ (\mathtt{s}\ x)))$, contradicting well-foundedness.

The dependency pair framework addresses both these issues. There are multiple higher-order definitions of dependency pairs, with distinct advantages and downsides; here, we present a form of *static* dependency pairs, both for its ease in presentation and because the static approach allows for more modular proofs than the alternative, *dynamic* style. To use static dependency pairs, we limit interest to *accessible function passing* (AFP) rules.

▶ **Definition 6.** *A set of rules $\mathcal{R}$ is* accessible function passing *if there exists a sort ordering $\succeq^{\mathbb{S}}$ such that: for all $\mathtt{f}\ \ell_1 \cdots \ell_k \to r \in \mathcal{R}$ and all $x \in \mathcal{FV}(r)$, there exists i with $\ell_i \trianglerighteq_{acc} x$.*

This requirement means that higher-order variables are used in an essentially harmless way. An example of a non-AFP rule is the encoding of the untyped λ-calculus: $\mathtt{app}\ (\mathtt{lam}\ F)\ X \to F\ X$, with $\mathtt{lam} :: (\mathsf{o} \Rightarrow \mathsf{o}) \Rightarrow \mathsf{o}$ and $\mathtt{app} :: \mathsf{o} \Rightarrow \mathsf{o} \Rightarrow \mathsf{o}$, where a higher-order variable is lifted out of a base-type term. There are also terminating systems which are not AFP. However, practical examples typically satisfy this requirement. For example, the rule $\mathtt{lapply}\ x\ (\mathtt{fcons}\ F\ a) \to F\ (\mathtt{lapply}\ x\ a)$ with $\mathtt{fcons} :: (\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{flist} \Rightarrow \mathsf{flist}$ also lifts a higher-order variable out of a base-type term, but is AFP if we choose $\mathsf{flist} \succ^{\mathbb{S}} \mathsf{nat}$.

In this paper, we will mostly consider rules $\mathtt{f}\ \ell_1 \cdots \ell_k \to r$ where all higher-order variables occur as a direct argument of the left-hand side (i.e., as one of the ℓ_i); this is the case for all rules in our running example. Such rules are AFP by letting $\succeq^{\mathbb{S}}$ equate all sorts.

▶ **Definition 7.** *For each defined symbol $\mathtt{f} :: \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota$, we introduce a fresh symbol $\mathtt{f}^\sharp :: \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \mathsf{dp}$. The set of* static dependency pairs *of $\mathcal{R}$ is given by: $\mathsf{SDP}(\mathcal{R}) = \{\mathtt{f}^\sharp\ \ell_1 \cdots \ell_k \Rrightarrow \mathtt{g}^\sharp\ r_1 \cdots r_n\ x_{n+1} \cdots x_m \mid \mathtt{f}\ \ell_1 \cdots \ell_k \to r \in \mathcal{R} \wedge r \trianglerighteq \mathtt{g}\ r_1 \cdots r_n \wedge \mathtt{g} \in \mathcal{D} \wedge \mathtt{g}\ r_1 \cdots r_n :: \sigma_{n+1} \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota \wedge x_{n+1} \in \mathcal{V}_{\sigma_1}, \ldots, x_m \in \mathcal{V}_{\sigma_m}$ are fresh variables$\}$.*

The set of static dependency pairs is obtained by taking, for each rule $\ell \to r$, all maximally applied subterms p of r headed by a defined symbol, if necessary applying p to fresh variables to obtain a base-type term, and marking the head symbols of both ℓ and p to indicate their special role. In the first order setting, dependency pairs trace function calls. In the (static) higher-order setting, they also trace *potential* calls: a call of function type might end up being applied to almost anything, which is represented by the fresh variables.

▶ **Example 8.** Our running example has the following dependency pairs:

A.	$\mathtt{inc}^\sharp\ \mathtt{0}$	$\Rrightarrow$	$\mathtt{inc}^\sharp\ (\mathtt{s}\ \mathtt{0})$	J.	$\mathtt{map}^\sharp\ F\ (\mathtt{cons}\ x\ a)$	$\Rrightarrow$	$\mathtt{map}^\sharp\ F\ a$
B.	$\mathtt{exp}^\sharp\ (\mathtt{s}\ x)\ y$	$\Rrightarrow$	$\mathtt{double}^\sharp\ x\ y\ \mathtt{0}$	K.	$\mathtt{fold}^\sharp\ F\ x\ (\mathtt{cons}\ y\ a)$	$\Rrightarrow$	$\mathtt{fold}^\sharp\ F\ (F\ x\ y)\ a$
C.	$\mathtt{min}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)$	$\Rrightarrow$	$\mathtt{min}^\sharp\ x\ y$	L.	$\mathtt{quot}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)$	$\Rrightarrow$	$\mathtt{quot}^\sharp\ (\mathtt{min}\ x\ y)\ (\mathtt{s}\ y)$
D.	$\mathtt{ack}^\sharp\ (\mathtt{s}\ x)\ \mathtt{0}$	$\Rrightarrow$	$\mathtt{ack}^\sharp\ x\ (\mathtt{s}\ \mathtt{0})$	M.	$\mathtt{quot}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)$	$\Rrightarrow$	$\mathtt{min}^\sharp\ x\ y$
E.	$\mathtt{ack}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)$	$\Rrightarrow$	$\mathtt{ack}^\sharp\ (\mathtt{s}\ x)\ y$	N.	$\mathtt{ack}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)$	$\Rrightarrow$	$\mathtt{ack}^\sharp\ x\ (\mathtt{ack}\ (\mathtt{s}\ x)\ y)$
F.	$\mathtt{double}^\sharp\ x\ \mathtt{0}\ z$	$\Rrightarrow$	$\mathtt{exp}^\sharp\ x\ z$	O.	$\mathtt{double}^\sharp\ x\ (\mathtt{s}\ y)\ z$	$\Rrightarrow$	$\mathtt{double}^\sharp\ x\ y\ (\mathtt{s}\ (\mathtt{s}\ z))$
G.	$\mathtt{mkbig}^\sharp\ a\ x$	$\Rrightarrow$	$\mathtt{ack}^\sharp\ x\ y$	P.	$\mathtt{mkbig}^\sharp\ a\ x$	$\Rrightarrow$	$\mathtt{map}^\sharp\ (\mathtt{ack}\ x)\ a$
H.	$\mathtt{mkdiv}^\sharp\ a\ x$	$\Rrightarrow$	$\mathtt{quot}^\sharp\ y\ x$	Q.	$\mathtt{mkdiv}^\sharp\ a\ x$	$\Rrightarrow$	$\mathtt{map}^\sharp\ (\lambda y.\mathtt{quot}\ y\ x)\ a$
I.	$\mathtt{sma}^\sharp\ \bot\ F\ (\mathtt{s}\ x)$	$\Rrightarrow$	$\mathtt{quot}^\sharp\ x\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0}))$	R.	$\mathtt{sma}^\sharp\ \bot\ F\ (\mathtt{s}\ x)$	$\Rrightarrow$	$\mathtt{sma}^\sharp\ (F\ x)\ F\ (\mathtt{quot}\ x\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0})))$

Note that DP (G), which came from the rule $\mathtt{mkbig}\ a\ x \to \mathtt{map}\ (\mathtt{ack}\ x)\ a$, has a fresh variable y in the right-hand side which does not occur on the left; this was used to flatten the subterm $\mathtt{ack}\ x$ to base type. (H) also has a variable y which occurs on the right but not the left; this is because the bound variable in $\mathtt{map}\ (\lambda y.\mathtt{quot}\ y\ x)\ a$ is freed in the subterm.

Dependency pairs are used by translating non-termination to absence of infinite *chains*:

▶ **Definition 9.** *For $\mathcal{P}$ a set of dependency pairs, and $\mathcal{R}$ a set of rules, a $(\mathcal{P}, \mathcal{R})$*-chain *is an infinite sequence $[(\ell_i \Rrightarrow r_i, \gamma_i) \mid i \in \mathbb{N}]$ such that for all i: $\ell_i \Rrightarrow r_i \in \mathcal{P}$, and $r_i\gamma_i \to_{\mathcal{R}}^* \ell_{i+1}\gamma_{i+1}$.*

A $(\mathcal{P}, \mathcal{R})$-chain is computable *if each $r_i\gamma_i$ is computable with respect to $\to_{\mathcal{R}}$.*

Essentially, a $(\mathcal{P}, \mathcal{R})$-chain represents an infinite reduction $s_1 \to_{\mathcal{P}} t_1 \to_{\mathcal{R}}^* s_2 \to_{\mathcal{P}} t_2 \to_{\mathcal{R}}^* s_3 \ldots \to_{\mathcal{P}}$, where each $s_i = \ell_i\gamma_i$ and $t_i = r_i\gamma_i$, and the steps using $\to_{\mathcal{P}}$ are at the root of s_i. Although chains can have various properties (e.g., being *minimal*, *computable*, *formative*), we here only consider *computability*, and only implicitly: this property – which implies that each $r_i\gamma_i$ is terminating, and that the immediate arguments of each $\ell_i\gamma_i$ are computable – is used in the (omitted) correctness proofs of Section 4. We have the following result:

▶ **Lemma 10.** *Let $\mathcal{R}$ be a set of accessible function passing rules (for a fixed sort ordering with* dp *maximal in $\succeq^{\mathbb{S}}$). If $\to_{\mathcal{R}}$ is non-terminating, then there is a computable $(\mathtt{SDP}(\mathcal{R}), \mathcal{R})$-chain.*

Hence, if we can prove that there is no such chain, we know the system terminates. One way of doing this is by using a well-founded ordering as before. Since the steps $s_i \to_{\mathcal{P}} t_i$ occur at the root of a term, it is not needed for $\succ$ to be monotonic. Rather, it suffices to use a *reduction pair*: a pair $(\succ, \succeq)$ that that $\succ$ is a well-founded ordering, $\succeq$ is a quasi-ordering, $\succ \cdot \succeq \subseteq \succ$, both relations are stable, $\succeq$ is monotonic, and $\to_\beta \subseteq \succeq$. We can again use interpretations to define a reduction pair. This is formally defined as follows:

▶ **Definition 11.** *We assume given, for all sorts ι, a well-founded set $(\mathcal{A}_\iota, \sqsupset_\iota, \sqsupseteq_\iota)$. This definition is extended to all simple types as follows: $\mathcal{A}_{\sigma \Rightarrow \tau} = \{f \in \mathcal{A}_\sigma \Longrightarrow \mathcal{A}_\tau \mid f$ is weakly monotonic$\}$; we let $\sqsupset_{\sigma\Rightarrow\tau}$ and $\sqsupseteq_{\sigma\Rightarrow\tau}$ denote the pointwise comparisons on these functions.*

For every $(\mathtt{f} :: \sigma) \in \mathcal{F}$, we assume given $\mathcal{J}_{\mathtt{f}} \in \mathcal{A}_\sigma$. For a closed term s let $[\![s]\!] = [\![s]\!]_\emptyset$, where, for α a function mapping each $(x :: \sigma) \in \mathcal{V} \cap \mathcal{FV}(s)$ to an element of $\mathcal{A}_\sigma$, we define:

$$\begin{array}{rclrcl} [\![\mathtt{f}]\!]_\alpha & = & \mathcal{J}_{\mathtt{f}} & [\![x]\!]_\alpha & = & \alpha(x) \\ [\![t\ u]\!]_\alpha & = & [\![t]\!]_\alpha([\![u]\!]_\alpha) & [\![\lambda x.t]\!]_\alpha & = & d \mapsto [\![t]\!]_{\alpha[x:=d]} \end{array}$$

Here, $\alpha[x := d]$ maps x to d and all other variables y to $\alpha(y)$, and $d \mapsto [\![t]\!]_{\alpha[x:=d]}$ is the function that maps $d \in \mathcal{A}_\sigma$, to $[\![t]\!]_{\alpha[x:=d]}$. If $s :: \sigma$, this definition yields an element $[\![s]\!]_\alpha$ of $\mathcal{A}_\sigma$. We will often omit the type denotations from $\sqsupseteq$ when they are clear from context or irrelevant. We will also usually omit α and instead use for instance $[\![\mathtt{f}\ x]\!] = [\![x]\!] + 1$ instead of $[\![\mathtt{f}(x)]\!]_\alpha = \alpha(x) + 1$. We typically choose $[\![\cdot]\!]$ to represent a kind of size measure on terms.

▶ **Example 12.** Let $\mathcal{A}_{\mathsf{list}} = \mathbb{N}$, ordered as usual. To prove that there is no $(\mathtt{SDP}(\mathcal{R}_{\mathtt{map}}), \mathcal{R}_{\mathtt{map}})$-chain, it suffices to find an interpretation function $\mathcal{J}$ with:

$$[\![\mathtt{map}\ F\ \mathtt{nil}]\!] \ \geq\ [\![\mathtt{nil}]\!] \qquad [\![\mathtt{map}\ F\ (\mathtt{cons}\ H\ T)]\!] \ \geq\ [\![\mathtt{cons}\ (F\ H)\ (\mathtt{map}\ F\ T)]\!]$$
$$[\![\mathtt{map}^\sharp\ F\ (\mathtt{cons}\ H\ T)]\!] \ >\ [\![\mathtt{map}^\sharp\ F\ T]\!]$$

This is easily accomplished by choosing $\mathcal{J}_{\mathtt{nil}} = 0$, $\mathcal{J}_{\mathtt{cons}}(x, y) = y + 1$, $\mathcal{J}_{\mathtt{map}}(F, y) = \mathcal{J}_{\mathtt{map}^\sharp}(F, y) = y$; that is, we map a term of list type to the length of the list. Then the above inequalities evaluate to: $0 \geq 0$, $T + 1 \geq T + 1$ and $T + 1 > T$.

Note that there is no obligation to choose $\mathcal{A}_\iota = \mathbb{N}$ for all sorts. For more complex systems than `map`, it may also be useful to for instance map sorts to the rational numbers, or to sets of terminating terms. In Section 5, we will map sorts to *tuples* of (natural) numbers.

As we have seen, dependency pairs and weakly monotonic interpretations together provide a method to prove termination. However, in contrast to the DP approach in first-order term rewriting, this is not a complete method: there are terminating systems which admit a computable chain (for example, $\mathcal{R} = \{\mathtt{f\ a} \to \mathtt{g\ f}\}$, which has a dependency pair $\mathtt{f\ a} \Rrightarrow \mathtt{f\ X}$). Hence, the method in general cannot be used for non-termination, and also has important limitations in its applicability for termination, even beyond the restriction to AFP rules.

The alternative, *dynamic* style of dependency pairs[16], does not come with applicability restrictions and does offer an if-and-only-if result. There, *collapsing* dependency pairs, of a form such as $\mathtt{map}^\sharp\ F\ (\mathtt{cons}\ H\ T) \Rrightarrow\ F\ H$, are included, and the notion of a $(\mathcal{P}, \mathcal{R})$-chain is somewhat more complex to support this. Unfortunately, this style is much worse at enabling modular proofs. That is why this paper focuses on the static approach.

4 Modular proofs with dependency pairs

The dependency pair framework allows "DP problems" to be progressively modified to prove absence of chains with certain properties. We here present a very simple version of this framework, which only modifies a set $\mathcal{P}$. A more elaborate framework is discussed in [11].

We fix an AFP set $\mathcal{R}$ of rules. Let a set $\mathcal{P}$ of DPs be called *chain-free* if there is no computable $(\mathcal{P}, \mathcal{R})$-chain. Then Lemma 10 states that $\to_{\mathcal{R}}$ is terminating if $\mathtt{SDP}(\mathcal{R})$ is chain-free. As suggested before, sets $\mathcal{P}$ can be simplified using a reduction pair. Formally:

▶ **Lemma 13.** *A set $\mathcal{P}$ is chain-free if $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$ where $\mathcal{P}_2$ is chain-free, and there is a reduction pair $(\succ, \succeq)$ such that: (a) $\ell \succ r$ for all $\ell \Rrightarrow r \in \mathcal{P}_1$, (b) $\ell \succeq r$ for all $\ell \Rrightarrow r \in \mathcal{P}_2$ and (c) $\ell \succeq r$ for all $\ell \to r \in \mathcal{R}$.*

Hence, chain-freeness of $\mathcal{P}$ is reduced to chain-freeness of a smaller set. Since $\succ$ does not need to be monotonic, it is often easier to remove a dependency pair in this way than it would be to remove a rule in the original system using rule removal.

▶ **Example 14.** Let $\mathcal{R} := \mathcal{R}_{\mathtt{quot}} \cup \mathcal{R}_{\mathtt{min}} \cup \{\mathtt{inc\ 0} \to \mathtt{inc\ (s\ 0)}\}$. Then $\mathcal{P} := \mathtt{SDP}(\mathcal{R})$ is the set $\{(\textsc{a}),(\textsc{c}),(\textsc{l}),(\textsc{m})\}$. We choose $\mathcal{J}$ to have $[\![\mathtt{0}]\!] = 0$, $[\![\mathtt{s}\ x]\!] = [\![x]\!] + 1$, $[\![\mathtt{inc}\ x]\!] = [\![\mathtt{inc}^\sharp\ x]\!] = 0$ and $[\![\mathtt{min}\ x\ y]\!] = [\![\mathtt{min}^\sharp\ x\ y]\!] = [\![\mathtt{quot}\ x\ y]\!] = [\![\mathtt{quot}^\sharp\ x\ y]\!] = [\![x]\!]$. Then $[\![\ell]\!] \geq [\![r]\!]$ for all $\ell \to r \in \mathcal{R}$, and moreover: each of (C), (L) and (M) reduces to $[\![\ell]\!] = x + 1 > x = [\![r]\!]$, while for (A) we have: $[\![\ell]\!] = 0 = [\![r]\!]$. By Lemma 13, we have chain-freeness of $\mathtt{SDP}(\mathcal{R})$ (and therefore termination of $\rightarrow_{\mathcal{R}}$) if we can prove chain-freeness of $\{\mathtt{inc}^\sharp\ \mathtt{0} \Rrightarrow \mathtt{inc}^\sharp\ (\mathtt{s}\ \mathtt{0})\}$. We avoid the problem noted in Example 5 because we only needed a *weakly* monotonic ordering.

While this is an improvement over using interpretations directly, it does nothing towards our goal: like with rule removal, in the first step we have to orient all the rules and dependency pairs in one go. Even though this is easier than before because $\succ$ does not need to be monotonic, it is still likely to be infeasible to handle thousands of rules at once.

So, let us consider an approach that does *not* need an ordering: the *splitting lemma.*

▶ **Lemma 15.** *Assume given disjoint sets of terms $A_1, \dots, A_n$, and suppose we can write $\mathcal{P} = \mathcal{P}_1 \cup \dots \cup \mathcal{P}_n \cup \mathcal{Q}_1 \cup \dots \cup \mathcal{Q}_n$ such that for all $i \in \{1, \dots, n\}$ we have:*

- *for all $\ell \Rrightarrow r \in \mathcal{P}_i \cup \mathcal{Q}_i$, and all substitutions γ: $\ell\gamma \in A_i$;*
- *for all $\ell \Rrightarrow r \in \mathcal{P}_i$, all substitutions γ and all terms s with $r\gamma \rightarrow_{\mathcal{R}}^* s$: $s \notin A_1 \cup \dots \cup A_{i-1}$;*
- *for all $\ell \Rrightarrow r \in \mathcal{Q}_i$, all substitutions γ and all terms s with $r\gamma \rightarrow_{\mathcal{R}}^* s$: $s \notin A_1 \cup \dots \cup A_i$.*

Then $\mathcal{P}$ is chain-free if and only if $\mathcal{P}_1, \dots, \mathcal{P}_n$ are all chain-free.

Note that the dependency pairs in $\mathcal{Q}_1 \cup \dots \cup Q_n$ are thrown away, while the others are split over potentially many smaller sets of dependency pairs that are truly interdependent. Essentially, this lemma is a different presentation of the *DP graph processor* [2, 12, 19].

▶ **Example 16.** Let $X^{\mathtt{f}}$ denote the set $\{\mathtt{f}^\sharp\ s_1 \cdots s_m \mid (\mathtt{f} :: \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \iota) \in \mathcal{F} \wedge s_1 :: \sigma_1, \dots, s_m :: \sigma_m\}$, so the set of all base-type terms s with $\mathtt{f}^\sharp$ as the head symbol.

For $\mathcal{R}$ the rules of Example 2, and $\mathcal{P} = \mathtt{SDP}(\mathcal{R})$ following Example 8, we may choose:

$$\begin{array}{lllll} A_1 := X^{\mathtt{mkbig}} & A_3 := X^{\mathtt{map}} & A_5 := X^{\mathtt{sma}} & A_7 := X^{\mathtt{min}} & A_9 := X^{\mathtt{double}} \cup X^{\mathtt{exp}} \\ A_2 := X^{\mathtt{mkdiv}} & A_4 := X^{\mathtt{fold}} & A_6 := X^{\mathtt{quot}} & A_8 := X^{\mathtt{ack}} & A_{10} := \{\mathtt{inc}^\sharp\ \mathtt{0}\} \end{array}$$

$$\begin{array}{lllll} \mathcal{P}_1 := \emptyset & \mathcal{P}_3 := \{(\textsc{j})\} & \mathcal{P}_5 := \{(\textsc{r})\} & \mathcal{P}_7 := \{(\textsc{c})\} & \mathcal{P}_9 := \{(\textsc{b}), (\textsc{f}), (\textsc{o})\} \\ \mathcal{Q}_1 := \{(\textsc{g}), (\textsc{p})\} & \mathcal{Q}_3 := \emptyset & \mathcal{Q}_5 := \{(\textsc{i})\} & \mathcal{Q}_7 := \emptyset & \mathcal{Q}_9 := \emptyset \\ \mathcal{P}_2 := \emptyset & \mathcal{P}_4 := \{(\textsc{k})\} & \mathcal{P}_6 := \{(\textsc{l})\} & \mathcal{P}_8 := \{(\textsc{d}), (\textsc{e}), (\textsc{n})\} & \mathcal{P}_{10} := \emptyset \\ \mathcal{Q}_2 := \{(\textsc{h}), (\textsc{q})\} & \mathcal{Q}_4 := \emptyset & \mathcal{Q}_6 := \{(\textsc{m})\} & \mathcal{Q}_8 := \emptyset & \mathcal{Q}_{10} := \{(\textsc{a})\} \end{array}$$

Here, we use the property that symbols $\mathtt{f}^\sharp$ do not occur in $\mathcal{R}$, so if the right-hand of a dependency pair has the form $\mathtt{f}^\sharp\ \vec{r}$, then the same holds for each term that $(\mathtt{f}^\sharp\ \vec{r})\gamma$ reduces to. Hence, essentially, we have an ordering on the function symbols, and let $\mathcal{P}_i$ be the set of dependency pairs where both sides have a function symbol of the same weight, and $\mathcal{Q}_i$ those where the right-hand side has a smaller weight than the left. In A_{10} we also consider the shape of the argument: since $\mathtt{inc}^\sharp\ (\mathtt{s}\ \mathtt{0})$ does not reduce and is not in A_{10}, Lemma 15 allows us to discard (A). We can also discard (G), (P), (H), (Q), (I) and (M), and reduce chain-freeness of $(\mathtt{SDP}(\mathcal{R}), \mathcal{R})$ to chain-freeness of each of $\mathcal{P}_3$, $\mathcal{P}_4$, $\mathcal{P}_5$, $\mathcal{P}_6$, $\mathcal{P}_7$, $\mathcal{P}_8$ and $\mathcal{P}_9$.

Yet, this still does not really accomplish our goal: while Lemma 15 allows us to split a large set into potentially many small ones, a small set of DPs is not necessarily easy to handle. In particular, to use Lemma 13, we still need to orient all rules in $\mathcal{R}$ at once.

Fortunately, in many cases we can avoid an ordering altogether using the *subterm criterion*:

▶ **Lemma 17.** *Given a set of dependency pairs $\mathcal{P}$, and a function π that maps each marked symbol $\mathtt{f}^\sharp :: \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \mathtt{dp}$ that occurs in $\mathcal{P}$ to an integer between 1 and m, let $\overline{\pi}(\mathtt{f}^\sharp\ s_1 \cdots s_m) := s_{\pi(\mathtt{f}^\sharp)}$. Suppose $\mathcal{P} = \mathcal{P}_= \cup \mathcal{P}_\rhd$, where $\overline{\pi}(\ell) = \overline{\pi}(r)$ for all $\ell \Rrightarrow r \in \mathcal{P}_=$ and $\overline{\pi}(\ell) \rhd \overline{\pi}(r)$ for all $\ell \Rrightarrow r \in \mathcal{P}_\rhd$. Then $\mathcal{P}$ is chain-free if and only if $\mathcal{P}_=$ is chain-free.*

The subterm criterion allows us to discard many dependency pairs without even considering $\mathcal{R}$. This is possible because the "chain-free' notion considers computable chains, so in a $(\mathcal{P},\mathcal{R})$-chain, each $\overline{\pi}(\ell)\gamma$ and $\overline{\pi}(r)\gamma$ can be assumed to be terminating.

▶ **Example 18.** Chain-freeness of $\{(\text{J})\}$ follows by $\pi(\mathtt{map}^\sharp) = 2$, since $\overline{\pi}(\mathtt{map}^\sharp\ F\ (\mathtt{cons}\ x\ a)) = \mathtt{cons}\ x\ a \rhd a = \overline{\pi}(\mathtt{map}^\sharp\ F\ a)$; we have $\mathcal{P}_= = \emptyset$ and $\mathcal{P}_\rhd = \{(\text{J})\}$, and $\emptyset$ is obviously chain-free. In the same way, $\{(\text{K})\}$ and $\{(\text{C})\}$ are discarded (choosing $\pi(\mathtt{fold}^\sharp) = 3$ for the first, and $\pi(\mathtt{min}^\sharp) = 1$ for the second). For the set $\{(\text{D}),(\text{E}),(\text{N})\}$, we let $\pi(\mathtt{ack}^\sharp) = 1$, and obtain chain-freeness if $\{(\text{E})\}$ is chain-free, which holds by a second application of the subterm criterion, now with $\pi(\mathtt{ack}^\sharp) = 2$. For $\{(\text{B}),(\text{F}),(\text{O})\}$, we let $\pi(\mathtt{exp}^\sharp) = \pi(\mathtt{double}^\sharp) = 1$, which allows us to discard (B) because $\mathtt{s}\ x \rhd x$; chain-freeness of the remaining set $\{(\text{F}),(\text{O})\}$ follows from chain-freeness of $\{(\text{O})\}$ by the splitting lemma (choosing $A_1 = X^{\mathtt{double}}$ and $A_2 = X^{\mathtt{exp}}$ as in Example 16), which follows by the subterm criterion with $\pi(\mathtt{double}^\sharp) = 2$.

Hence, following Example 16, Example 2 is terminating if $\{(\text{L})\}$ and $\{(\text{R})\}$ are chain-free.

The formulation and use of the subterm criterion is exactly as in the first-order case. There is a also variation of this criterion with a higher-order focus[11, Theorem 63]:

▶ **Lemma 19.** *Let $s \sqsupset t$ if $s \rhd_{acc} t$ or $t = F\ t_1 \cdots t_n$ and $s \rhd_{acc} F$ with $F \in \mathcal{V}$. $\mathcal{P}_= \cup \mathcal{P}_\rhd$ is chain-free if $\mathcal{P}_\rhd$ is chain-free, $\overline{\pi}(\ell) = \overline{\pi}(r)$ for $\ell \Rrightarrow r \in \mathcal{P}_=$ and $\overline{\pi}(\ell) \sqsupset \overline{\pi}(r)$ for $\ell \Rrightarrow r \in \mathcal{P}_\rhd$.*

So, the $\rhd$ relation in Lemma 17 is replaced by a relation that considers the type ordering and accessibility relation. This is designed particularly to handle rules like ordinal recursion: $\mathtt{rec}\ (\mathtt{lim}\ F)\ U\ X\ W \to W\ F\ (\lambda n.\mathtt{rec}\ (F\ n)\ U\ X\ W)$, which has a dependency pair $\mathtt{rec}^\sharp\ (\mathtt{lim}\ F)\ U\ X\ W \Rrightarrow \mathtt{rec}^\sharp\ (F\ n)\ U\ X\ W$ with $\mathtt{lim} :: (\mathsf{nat} \Rightarrow \mathsf{ord}) \Rightarrow \mathsf{ord}$.

The subterm criterion (whether in its basic form or the variation of Lemma 19) is a powerful technique that – in combination with the splitting lemma (Lemma 15) – might allow us to complete a termination proof in a very modular way. Yet, if any DP problems remain which cannot be further split by either lemma, we will still have to orient all the rules. To deal with this issue, we again follow the first-order DP framework and apply *usable rules*.

▶ **Definition 20** (Usable Rules). *For Q a set of rules or dependency pairs, let $\mathtt{rhs}(Q)$ denote the set of terms occurring as the right-hand side of some rule/DP in Q. For a set T of terms, let $\mathtt{Use}(T,\mathcal{R})$ denote the set of those rules $\mathtt{f}\ \ell_1 \cdots \ell_k \to r$ in $\mathcal{R}$ such that:*

1. *there is a term $s \in T$ which has a (fully applied) subterm of the form $\mathtt{f}\ s_1 \cdots s_k$, or*
2. *there is a term $s \in T$ which has a subterm $x\ t_1 \cdots t_m$ with $x \in \mathcal{FV}(s)$ and $m > 0$.*

For a set of DPs $\mathcal{P}$, we let its set $\mathsf{UR}(\mathcal{P},\mathcal{R})$ of usable rules *be defined as the smallest set $U \subseteq \mathcal{R}$ such that $\mathtt{Use}(\mathtt{rhs}(\mathcal{P}),\mathcal{R}) \subseteq U$ and $\mathtt{Use}(\mathtt{rhs}(U),\mathcal{R}) \subseteq U$.*

Intuitively, a rule is considered usable if we may need it to rewrite relevant instances of some right-hand side of $\mathcal{P}$. For example, when rewriting a term $\mathtt{f}\ (\mathtt{quot}\ s\ t)$, we will likely need the `quot` rules, and their use introduces occurrences of `min`, which may also be relevant. However, the `fold` rules will only be used if `fold` already occurs in s or t.

▶ **Example 21.** For our running example, $\mathsf{UR}(\{(\text{L})\ \mathtt{quot}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) \Rrightarrow \mathtt{quot}^\sharp\ (\mathtt{min}\ x\ y)\ (\mathtt{s}\ y)\}, \mathcal{R}) = \mathcal{R}_{\mathtt{min}}$, since the only defined symbol occurring in the right-hand side is `min`, and the right-hand side of the two `min` rules contain no other defined symbols. Note that $\mathtt{quot}^\sharp$ is marked, and does not occur in $\mathcal{R}$, so the `quot` rules are not included. $\mathsf{UR}(\{(\text{R})\ \mathtt{sma}^\sharp\ \bot\ F\ (\mathtt{s}\ x) \Rrightarrow \mathtt{sma}^\sharp\ (F\ x)\ F\ (\mathtt{quot}\ x\ (\mathtt{s}\ (\mathtt{s}\ 0)))\}, \mathcal{R}) = \mathcal{R}$ due to the subterm $F\ x$ of the right-hand side.

Usable rules are best used in combination with a weakly monotonic ordering. In the following, let $\mathcal{C}_\epsilon$ be a set $\{\mathtt{pair}_\iota\ x\ y \to x,\ \mathtt{pair}_\iota\ x\ y \to y \mid \iota \in \mathbb{S}\}$ for fresh symbols $\mathtt{pair}_\iota$.

▶ **Lemma 22.** *Suppose $\mathcal{R}$ is finitely branching. Then a set $\mathcal{P}$ is chain-free if $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$ where $\mathcal{P}_2$ is chain-free, and there is a reduction pair $(\succ, \succeq)$ such that: (a) $\ell \succ r$ for all $\ell \Rrightarrow r \in \mathcal{P}_1$, (b) $\ell \succeq r$ for all $\ell \Rrightarrow r \in \mathcal{P}_2$ and (c) $\ell \succeq r$ for all $\ell \to r \in \mathsf{UR}(\mathcal{P}, \mathcal{R}) \cup \mathcal{C}_\epsilon$.*

("Finitely branching" means that for any s there are only finitely many t with $s \to_{\mathcal{R}} t$; this holds for instance if $\mathcal{R}$ is finite.)

The difference between Lemma 22 and Lemma 13 is that instead of orienting all rules, we only have to orient the usable rules, plus some rules of the form $\mathtt{pair}_\iota\ x_1\ x_2 \to x_i$. The latter is trivial for most commonly used orderings. The need for these additional rules is also present in the first-order case, and can be dropped when considering *innermost* termination.

▶ **Example 23.** To prove chain-freeness of $\{(\text{L})\ \mathtt{quot}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) \Rrightarrow \mathtt{quot}^\sharp\ (\mathtt{min}\ x\ y)\ (\mathtt{s}\ y)\}$, whose DPs are $\mathcal{R}_{\mathtt{min}}$ following Example 21, we need $\mathtt{quot}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) \succ \mathtt{quot}^\sharp\ (\mathtt{min}\ x\ y)\ (\mathtt{s}\ y)$ and $\mathtt{min}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) \succeq \mathtt{min}\ x\ y$ and $\mathtt{min}\ x\ \mathtt{0} \succeq x$, as well as $\mathtt{pair}_\iota \succeq \iota$ for all ι. To achieve this, we use the same interpretation as in Example 14, and let $\mathcal{J}_{\mathtt{pair}_\iota} = \max(x, y)$ for all ι.

We have now nearly completed our running example, with only one singular set remaining. To address this last dependency pair, we observe that the use of the function symbol in the $\mathtt{sma}$ rules is innocuous: the size of $\mathtt{sma}\ b\ F\ x$ is bounded by the size of x no matter what kinds of calls the evaluation of F may bring up. It would be nice to ignore the dependency pairs imposed by this relatively harmless function application. To do this, we build on first-order methods once more, and combine usable rules with an *argument filtering.*

▶ **Definition 24** (Argument filtering). *Let a function ν be given which maps each (marked or unmarked) function symbol $\mathtt{f} :: \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota$ to a subset of $\{1, \ldots, m\}$. If $\nu(\mathtt{f}) = \{i_1, \ldots, i_k\}$ with $i_1 < \cdots < i_k$, then let $\psi_\nu(\mathtt{f}\ s_1 \cdots s_m)$ denote $\mathtt{f}'\ s_{i_1} \cdots s_{i_k}$, where $\mathtt{f}' :: \sigma_{i_1} \Rightarrow \ldots \Rightarrow \sigma_{i_k} \Rightarrow \iota$ is a new function symbol. We define:*

$$\begin{array}{rcl} \overline{\nu}(\mathtt{f}\ t_1 \cdots t_n) & = & \lambda x_{n+1} \ldots x_m.\psi_\nu(\mathtt{f}\ \overline{\nu}(t_1) \cdots \overline{\nu}(t_n)\ x_{n+1} \cdots x_m) \text{ if } \mathtt{f} \text{ takes } m \text{ args} \\ \overline{\nu}(x\ t_1 \cdots t_n) & = & x\ \overline{\nu}(t_1) \cdots \overline{\nu}(t_n) \\ \overline{\nu}((\lambda x.u)\ t_1 \cdots t_n) & = & (\lambda x.\overline{\nu}(u))\ \overline{\nu}(t_1) \cdots \overline{\nu}(t_n) \end{array}$$

For a set of rules $\mathcal{R}$, let $\overline{\nu}(\mathcal{R}) = \{\overline{\nu}(\ell) \to \overline{\nu}(r) \mid \ell \to r \in \mathcal{R}\}$, and similar for a set of DPs.

Essentially, we make sure that all function symbols are maximally applied (by replacing a partially applied function $\mathtt{f}\ s_1 \cdots s_n$ by $\lambda x_{n+1} \ldots x_m.\mathtt{f}\ s_1 \cdots s_n\ x_{n+1} \cdots x_m$), and then remove the arguments that we do not want to consider from their function symbols.

▶ **Lemma 25.** *Suppose $\mathcal{R}$ is finitely branching. Then a set $\mathcal{P}$ is chain-free if $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$ where $\mathcal{P}_2$ is chain-free, and there is a reduction pair $(\succ, \succeq)$ such that: (a) $\ell \succ r$ for all $\ell \Rrightarrow r \in \overline{\nu}(\mathcal{P}_1)$, (b) $\ell \succeq r$ for all $\ell \Rrightarrow r \in \overline{\nu}(\mathcal{P}_2)$ and (c) $\ell \succeq r$ for all $\ell \to r \in \mathsf{UR}(\overline{\nu}(\mathcal{P}), \overline{\nu}(\mathcal{R})) \cup \mathcal{C}_\epsilon$.*

With this method, we can finally complete our running example.

▶ **Example 26.** We let $\overline{\nu}(\mathtt{sma}^\sharp) = \{2, 3\}$ and $\overline{\nu}(\mathtt{f}) = \{1, \ldots, m\}$ for all other symbols $\mathtt{f} :: \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota$. Then $\overline{\nu}(\{(\text{R})\}) = \{\mathtt{sma}^\sharp\ F\ (\mathtt{s}\ x) \Rrightarrow \mathtt{sma}^\sharp\ F\ (\mathtt{quot}\ x\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0})))\}$. Hence, $\mathsf{UR}(\overline{\nu}(\{(\text{R})\}), \overline{\nu}(\mathcal{R})) = \mathsf{UR}(\overline{\nu}(\{(\text{R})\}), \mathcal{R}) = \mathcal{R}_{\mathtt{quot}} \cup \mathcal{R}_{\mathtt{min}}$.

We use the same interpretation for $\mathtt{quot}$ and $\mathtt{min}$ as in Example 14, and let $[\![\mathtt{sma}^\sharp\ F\ x]\!] = [\![x]\!]$. Then $[\![\ell]\!] \geq [\![r]\!]$ is satisfied for the usable rules as before, and $[\![\mathtt{sma}^\sharp\ F\ (\mathtt{s}\ x)]\!] = [\![x]\!] + 1 > [\![x]\!] = [\![\mathtt{sma}^\sharp\ F\ (\mathtt{quot}\ x\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0})))]\!]$ orients the DP. Hence, our last remaining set $\mathcal{P}$ is chain-free, and the original system is terminating.

In the context of step-wise simplifying a termination problem, *formative rules* are also worth mentioning. These are defined much like usable rules, but from the *left* side of rules and DPs rather than the *right*: $\mathtt{Form}(T, \mathcal{R})$ contains those $\ell \to r \in \mathcal{R}$ such that:

1. $r = \mathtt{f}\ r_1 \cdots r_m$ and there is a term $s \in T$ with $s \trianglerighteq \mathtt{f}\ s_1 \cdots s_m$ for some $s_1, \dots, s_m$, or
2. $r = x\ r_1 \cdots r_m$ and there is a term $s \in T$ with $s \trianglerighteq t$ for some t whose type is the same as the type of r, and t is not a free variable in s, or
3. there is a term $s \in T$ which is not linear, or has a subterm $\lambda x.t$ with $\mathcal{FV}(t) \cap \mathcal{FV}(s) \neq \emptyset$.

The set $\mathsf{FR}(\mathcal{P}, \mathcal{R})$ of formative rules is the smallest set $O \subseteq \mathcal{R}$ such that $\mathtt{Form}(\mathtt{lhs}(\mathcal{P}), \mathcal{R}) \subseteq O$ and $\mathtt{Form}(\mathtt{lhs}(O), \mathcal{R}) \subseteq O$. Hence, the parallels with usable rules are obvious.

In a more elaborate DP framework, which carries pairs $(\mathcal{P}, \mathcal{R})$ instead of just sets $\mathcal{P}$ and considers more properties for chains than just computability, this definition can be used to remove elements of $\mathcal{R}$ [11, Theorem 58]. In the current, limited DP framework, we can still use formative rules with reduction pairs, for instance by changing requirement (c) in Lemma 25 to: $\ell \succeq r$ for all $\ell \to r \in \mathsf{UR}(\overline{\nu}(\mathcal{P}), \overline{\nu}(\mathsf{FR}(\mathcal{P}, \mathcal{R}))) \cup \mathcal{C}_\epsilon$. It seems likely that we can also combine formative rules with an argument filtering, and hence limit interest to $\ell \to r \in \mathsf{UR}(\overline{\nu}(\mathcal{P}), \mathsf{FR}(\overline{\nu}(\mathcal{P}), \overline{\nu}(\mathcal{R}))) \cup \mathcal{C}_\epsilon$. However, this proof currently only exists as a sketch.

Unfortunately, although we can use this method to eliminate some rules, these rules are usually simple; for example, we may throw out the base case of a rule $\mathtt{times}\ \mathtt{0}\ y \to \mathtt{0}$ but not the more complex induction case $\mathtt{times}\ (\mathtt{s}\ x)\ y \to \mathtt{add}\ (\mathtt{s}\ x)\ (\mathtt{times}\ x\ y)$. The primary use case is when the set of sorts can be split, say $\mathbb{S} = A \cup B$, so that the rules of type A do not use any symbols over type B; in this case, we may be able to remove all rules of type B. However, this does not happen often in practice. Hence, this is not really a core technique.

Discussion. The techniques in this section are all direct adaptations of methods for first-order term rewriting, and they are used in a similar way as their first-order counterpart. Yet, there is a clear place for higher-order reasoning, too. Type analysis play a role in both the AFP restriction and the alternative subterm criterion. In the splitting lemma, higher-order reachability analysis can be used to assess whether any reducts of $r\gamma$ are in some A_i. The choice of a reduction pair needs to take functional variables and β-reduction into account.

A critical difference between first-order and higher-order analysis lies in usable rules: case 2 in Definition 20 is not present in the first-order definition, since there variables cannot be applied. But in higher-order rewriting, if any element of $\mathcal{P}$, *or any of its usable rules*, has a subterm $x\ s_0 \cdots s_n$, then all rules are usable. Since a variable of higher type is typically applied eventually (otherwise, why carry it around?), this essentially means that if any rule with a higher-order variable is usable, then all rules are, and Lemma 22 is no improvement over Lemma 13. Effectively: we can only use usable rules in an essentially first-order problem!

Hence, instead of usable rules, Example 23 could have been done using [9], which shows that if the "first-order" part of a higher-order system combined with $\mathcal{C}_\epsilon$ is terminating, then the corresponding DPs may be dropped from $\mathtt{SDP}(\mathcal{R})$. We recover this result with Lemmas 15 and 22: define FO as the largest subset of $\mathcal{R}$ such that (a) the rules in FO do not use abstractions, variables of higher type or partially applied function symbols, and (b) $\mathtt{Use}(\mathtt{rhs}(\mathsf{FO}), \mathcal{R}) \subseteq \mathsf{FO}$. Let $A_2 = \{\mathtt{f}^\sharp\ s_1 \cdots s_n \mid \mathtt{f}$ is the head symbol of the left-hand side of a rule in $\mathsf{FO}\}$, and let $A_1 = \{\mathtt{f}^\sharp\ s_1 \cdots s_m \mid \mathtt{f}$ is a different defined symbol$\}$; by Lemma 15, termination follows if $\mathtt{SDP}(\mathcal{R} \setminus \mathsf{FO})$ and $\mathtt{SDP}(\mathsf{FO})$ are both chain-free. As the usable rules of $\mathtt{SDP}(\mathsf{FO})$ are in FO, we can apply Lemma 22 with $\succ$ the (terminating!) relation $(\to_{\mathsf{FO} \cup \mathcal{C}_\epsilon} \cup \triangleright)^+$ on terms with $\sharp$ marks removed. Hence, it suffices to prove chain-freeness of $\mathtt{SDP}(\mathcal{R} \setminus \mathsf{FO})$.

A similar result appears in [13], but instead of just first-order rules, this paper considers a set $A \subseteq \mathcal{R}$ where both the left- and right-hand sides of rules are patterns. This obviously captures first-order rules, but – due to the more permissive formalism of rewriting used in [13] – also some forms of higher-order rules with particular applications (algebraic effect handlers). To handle $\mathcal{R} \setminus A$, the author of [13] does not use dependency pairs but rather a version of the general schema [4]. There are many similarities between this technique and dependency pairs with the splitting lemma and extended subterm criterion, but the restrictions to apply the general schema do *not* need to apply to A. A parallel result in our setting would be that the rules of A would not need to be accessible function passing, yet termination still holds if $\mathtt{SDP}(\mathcal{R} \setminus A)$ is chain-free. It might be worth investigating if this is the case.

These positive results aside, without an argument filtering, usable rules does not give us much else due to the requirement that any variable application makes all rules usable. Unfortunately, this requirement is hard to avoid. Consider for instance the rules $\mathcal{R}_{\mathtt{comp2}}$:

$$\begin{array}{rclrcl} \mathtt{comp2}\ \mathtt{0}\ (\mathtt{s}\ y) & \to & \bot & \mathtt{comp2}\ x\ \mathtt{0} & \to & \top \\ \mathtt{comp2}\ (\mathtt{s}\ \mathtt{0})\ (\mathtt{s}\ y) & \to & \bot & \mathtt{comp2}\ (\mathtt{s}\ (\mathtt{s}\ x))\ (\mathtt{s}\ y) & \to & \mathtt{comp2}\ x\ y \\ \mathtt{f}\ F\ x\ \bot & \to & \mathtt{end}\ x & \mathtt{f}\ F\ x\ \top & \to & \mathtt{f}\ F\ (\mathtt{s}\ x)\ (\mathtt{comp2}\ (F\ x)\ x) \end{array}$$

Now, $\to_{\mathcal{R}_{\mathtt{comp2}} \cup \mathcal{C}_\epsilon}$ is terminating, since $\mathtt{comp2}\ n\ m$ determines whether $n \geq 2 * m$, and the only closed functions from $\mathtt{nat}$ to $\mathtt{nat}$ are built using λ, $\mathtt{0}$, $\mathtt{s}$ and $\mathtt{pair}_{\mathtt{nat}}$. Hence, in the worst case F is linear in its argument, so for large enough x, $\mathtt{comp2}\ (F\ x)\ x$ will return $\bot$. However, combining these rules with $\mathtt{double}\ \mathtt{0} \to \mathtt{0}$, $\mathtt{double}\ (\mathtt{s}\ x) \to \mathtt{s}\ (\mathtt{s}\ (\mathtt{double}\ x))$ clearly yields a non-terminating system. Here it is essential that the $\mathtt{double}$ rules are considered usable.

All this means that, if we succeed in applying usable rules – with or without an argument filtering – the corresponding ordering requirements will be essentially first-order (perhaps with some abstractions or unused higher-order variables). When these methods do not apply, there is no obvious way to circumvent the need to orient all rules at once. The same happens when we use *dynamic* instead of *static* DPs, where collapsing pairs often cause the subterm criterion, splitting lemma and usable rules to fail; the static approach is incomplete, so we may need the dynamic approach even on some AFP systems. In the next section we will see how we can also use a modular kind of reasoning to build a suitable reduction pair.

5 Incrementally building weakly monotonic interpretations

Although higher-order variations of the *recursive path ordering* [14, 5] have been very succesful in orienting higher-order rules, the current paper instead focuses on *interpretations*. The reason for this is twofold. First, the static dependency pair approach already captures many of the same advantages as higher-order RPO, since both methods are based on the same proof technique (computability). The second, and main, reason is that, unlike RPO, an interpretation-based ordering for a large set of rules can usually be built step by step.

Weakly monotonic interpretations do not provide a complete proof method: there are terminating systems that cannot be ordered with interpretations. Nevertheless, it has the potential to be very powerful – if we choose the sets $\mathcal{A}_\iota$ right. In the examples so far, we have let $\mathcal{A}_\iota = \mathbb{N}$ for all sorts, but this is fundamentally limiting. For example, if other rules impose that $[\![\mathtt{s}\ x]\!] > [\![x]\!]$, we cannot orient $\mathtt{inc}\ \mathtt{0} \to \mathtt{s}\ (\mathtt{inc}\ (\mathtt{s}\ \mathtt{0}))$. Instead, following an approach for complexity in [17], we will map terms to *tuples* of numbers.

Intuitively, we assign to all sorts a variety of numbers to indicate different measures of *size*. For example, a string of as and bs might be mapped to the number of as, the number of bs, and the total length. Then we express for each rule how it affects the size measures. This is a semantic technique: rather than only looking at the shape of rules, the best results are typically obtained by modelling our interpretation to the intended meaning of the rules.

We left Section 4 with some techniques that *often*, but not *always* allow us to cut a termination proof into bite-sized chunks. In the remaning cases, we must orient a large number of rules and – typically – a small number of DPs using a reduction pair. To find an interpretation (following Definition 11) that lets us do so, we will use the following procedure:

1. We choose an initial set $\mathcal{A}_\iota$ for each sort, along with an intuitive meaning, and define $\mathcal{J}_{\mathtt{f}}$ for all constructor symbols `f` according to this meaning.
2. We divide the defined symbols into sets $\mathcal{D}_1, \dots, \mathcal{D}_n$ such that for each $\mathtt{f} \in \mathcal{D}_i$, all the function symbols occurring in the rules defining `f` are either constructors or in $\mathcal{D}_1 \cup \dots \cup \mathcal{D}_i$.
3. For all i (starting with 1 going up to n), we find interpretations for the symbols in $\mathcal{D}_i$ so that $[\![\ell]\!] \sqsupseteq [\![r]\!]$; we strive to make them *as tight as possible*, to make later rules easier.
4. If we find that some rule of sort ι cannot be oriented, we extend $\mathcal{A}_\iota$ with an additional measure that does make this possible (if we can). We return to the previous step, updating the interpretations we already had to take the new measure into account.
5. When all rules are oriented, we find interpretations for the DPs in the same way.

This approach has not been formalised or implemented; rather, the goal is to present *ideas*; to hopefully lay the foundation for an automated approach in the future.

Let us explore how the procedure works by applying it to a large example.

Preparation. Let $\mathcal{R}$ consist of the rules in Example 2 combined with the following:

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{hd}\ (\mathtt{cons}\ x\ a) & \to & x & \mathtt{len}\ \mathtt{nil} & \to & \mathtt{0} \\ \mathtt{id}\ x & \to & x & \mathtt{len}\ (\mathtt{cons}\ x\ a) & \to & \mathtt{s}\ (\mathtt{len}\ a) \\ \mathtt{twice}\ F\ x & \to & F\ (F\ x) & \mathtt{H}\ (\mathtt{s}\ x) & \to & \mathtt{H}\ (\mathtt{twice}\ \mathtt{id}\ x) \end{array}$$

For $\mathcal{P} = \{\mathtt{H}^\sharp\ (\mathtt{s}\ x) \Rrightarrow \mathtt{H}^\sharp\ (\mathtt{twice}\ \mathtt{id}\ x)\} \subseteq \mathtt{SDP}(\mathcal{R})$, all rules are usable, the subterm criterion cannot be applied, and there is no argument filtering that stops all rules from being usable and yet allows us to strictly orient the single dependency pair. Hence, as we noted before, we need to find an interpretation to show $[\![\ell]\!] \succeq [\![r]\!]$ for a large number of rules (all rules in the system), and $[\![\ell]\!] \succ [\![r]\!]$ for a small number of DPs (the single element of $\mathcal{P}$).

So let us begin! Following step 1, we assign an intuitive measure to each type: terms of type `nat` are mapped to the corresponding number, lists to their largest element, and booleans to 0 or 1: $\mathcal{A}_{\mathsf{nat}} = \mathcal{A}_{\mathsf{list}} = (\mathbb{N}, >, \geq)$, $\mathcal{A}_{\mathsf{bool}} = (\{0,1\}, >, \geq)$. This corresponds with:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl} \mathcal{J}_{\mathsf{0}} & = & 0 & \mathcal{J}_{\mathtt{nil}} & = & 0 & \mathcal{J}_{\bot} & = & 0 \\ \mathcal{J}_{\mathsf{s}}(x) & = & x+1 & \mathcal{J}_{\mathsf{cons}}(x,a) & = & \max(x,a) & \mathcal{J}_{\top} & = & 1 \end{array}$$

We will handle the defined symbols in the following order: {`id`}, {`twice`}, {`min`}, {`quot`}, {`sma`}, {`hd`}, {`ack`}, {`map`}, {`mkbig`}, {`mkdiv`}, {`len`}, {`fold`}, {`inc`}, {`double`, `exp`}. This satisfies the requirement on the order of symbols, and is otherwise arbitrary.

The straightforward part. Following step 3, we will repeatedly interpret one or more defined symbols whose rules only depend on each other and symbols that already have an interpretation. To start, if $\mathcal{J}_{\mathtt{id}}(x) = x$ clearly $[\![\mathtt{id}\ x]\!] = [\![x]\!]$. The rule defining `id` is oriented, and since we have an equality, this interpretation is as tight as possible. We can achieve the same for `twice`: with $\mathcal{J}_{\mathtt{twice}}(F,x) = F(F(x))$ we have $[\![\ell]\!] = [\![r]\!]$ for the corresponding rule.

Unfortunately, we cannot achieve equality for `min`. Due to the monotonicity requirement, we cannot have $\mathcal{J}_{\mathtt{min}}(x,y) = x - y$, which would give a tight interpretation. For the current choice of $(\mathcal{A}_{\mathsf{nat}}, \sqsupset_{\mathsf{nat}}, \sqsupseteq_{\mathsf{nat}})$, the best we can do is $\mathcal{J}_{\mathtt{min}}(x,y) = x$. With this choice, $[\![\mathtt{min}\ x\ \mathtt{0}]\!] = [\![x]\!]$, and $[\![\mathtt{min}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)]\!] = [\![x]\!] + 1 > [\![x]\!] = [\![\mathtt{min}\ x\ y]\!]$, so the rules are oriented.

Next is `quot`. Since we already know $\mathcal{J}_{\mathtt{f}}$ for all other symbols in the two `quot` rules, the requirements are: $[\![\mathtt{quot}\ \mathtt{0}\ (\mathtt{s}\ y)]\!] = \mathcal{J}_{\mathsf{quot}}(0, y+1) \geq 0 = \mathcal{J}_{\mathsf{0}} = [\![\mathtt{0}]\!]$, and $[\![\mathtt{quot}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)]\!] = \mathcal{J}_{\mathsf{quot}}(x+1, y+1) \geq \mathcal{J}_{\mathsf{quot}}(x, y+1) + 1 = [\![\mathtt{s}\ (\mathtt{quot}\ (\mathtt{min}\ x\ y)\ (\mathtt{s}\ y))]\!]$. This is easily satisfied with $\mathcal{J}_{\mathsf{quot}}(x,y) = x$ (which is tight, as the left- and right-hand side are equal in both rules).

Similarly, the requirements for sma are: $\mathcal{J}_{\mathtt{sma}}(b, F, 0) \geq 0$ and $\mathcal{J}_{\mathtt{sma}}(1, F, x+1) \geq x+1$ and $\mathcal{J}_{\mathtt{sma}}(0, F, x+1) \geq \mathcal{J}_{\mathtt{sma}}(F(x), F, x)$. The simplest solution is $\mathcal{J}_{\mathtt{sma}}(b, F, x) = x$. To orient $\mathtt{hd}\ (\mathtt{cons}\ x\ a) \to x$, we let $\mathcal{J}_{\mathtt{hd}}(x) = x$; this suffices because $\max(x, a) \geq x$, and is optimal.

Beyond polynomials. When adressing ack, we run into some trouble: thus far, all our interpretation functions $\mathcal{J}_{\mathtt{f}}$ have been bounded by polynomials, but these rules implement the Ackermann function which grows much faster than any polynomial. However, there is no need to limit interest to polynomials. Indeed, the three rules provide a recursive specification:

$$\begin{array}{rclrcl} \mathtt{ack}\ 0\ y & = & \mathtt{s}\ y & \mathtt{ack}\ (\mathtt{s}\ x)\ 0 & = & \mathtt{ack}\ x\ (\mathtt{s}\ 0) \\ & & & \mathtt{ack}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) & = & \mathtt{ack}\ x\ (\mathtt{ack}\ (\mathtt{s}\ x)\ y) \end{array}$$

We can see by the recursive path ordering that this is terminating, and since it is a non-overlapping constructor system, it is confluent. Hence, we can define *Ack* as a function from $\mathbb{N}$ to $\mathbb{N}$, and choose $\mathcal{J}_{\mathtt{ack}}(x, y) = Ack(x, y)$. Then obviously all three ack rules are oriented.

We orient map by $\mathcal{J}_{\mathtt{map}}(F, a) = F(a)$: by weak monotonicity of F we have $F(\max(x, a)) \geq F(x)$. Intuitively, applying F to *some* element of the list cannot be greater than F(largest element). To orient the mkbig rules, we must have $\mathcal{J}_{\mathtt{mkbig}}(a, x) \geq \mathcal{J}_{\mathtt{map}}(\mathcal{J}_{\mathtt{ack}}(x), a) = Ack(x, a)$, so we choose $\mathtt{mkbig}(a, x) = Ack(x, a)$. For mkdiv, we let $\mathcal{J}_{\mathtt{mkdiv}}(x, a) = \mathcal{J}_{\mathtt{quot}}(a, x) = a$.

Backtracking. We are in trouble again when trying to orient the len rule: the interpretation of the constructors imposes $\mathcal{J}_{\mathtt{len}}(0) = 0$ and $\mathcal{J}_{\mathtt{len}}(\max(x, a)) \geq 1 + \mathcal{J}_{\mathtt{len}}(a)$. The latter is not satisfiable since (for $x = a$) it implies $\mathcal{J}_{\mathtt{len}}(a) \geq 1 + \mathcal{J}_{\mathtt{len}}(a)$. The problem lies in the choice for $\mathcal{J}_{\mathtt{cons}}$, which does not give enough information. Similarly, if we had chosen $\mathcal{J}_{\mathtt{cons}}(x, a) = a + 1$ (so mapping a list to its length), we could have oriented the len rules but not hd.

Hence, we are at Step 4: extending the sort interpretations. We can keep $\mathcal{A}_{\mathsf{nat}}$ unchanged, but let us take $\mathcal{A}_{\mathsf{list}} := \mathbb{N}^2$, mapping a list of numbers to the pair of its greatest argument and its length (ordered with $\geq^{\times}$ as described in Section 2). The constructors are mapped to:

$$\mathcal{J}_{\mathtt{nil}} = \langle 0, 0 \rangle \qquad \mathcal{J}_{\mathtt{cons}}(x, \langle m, l \rangle) = \langle \max(x, m), l + 1 \rangle$$

This follows the intended meaning of the sort. In line with Step 4 we now need to go back and update all interpretations for the new target set $\mathcal{A}_{\mathsf{nat}}$ and the new interpretations for nil and cons. However, this turns out to be quite easy. Note that in the interpretations of the constructors, the original choices 0 and $\max(x, a))$ are still present, in the first component. Similarly, the interpretations for the defined symbols are adapted by (a) replacing any list variable by its first component, and (b) adding a length component to the interpretation for the defined symbols of a type $\vec{\sigma} \Rightarrow \mathsf{list}$, so that $[\![\ell]\!]_2 \geq [\![r]\!]_2$ for the relevant rules. This yields:

Original:	**Update:**
$\mathcal{J}_{\mathtt{hd}}(a) = a$	$\mathcal{J}_{\mathtt{hd}}(\langle m, l \rangle) = m$
$\mathcal{J}_{\mathtt{map}}(F, a) = F(a)$	$\mathcal{J}_{\mathtt{map}}(F, \langle m, l \rangle) = \langle F(m), l \rangle$
$\mathcal{J}_{\mathtt{mkbig}}(a, x) = Ack(x, a)$	$\mathcal{J}_{\mathtt{mkbig}}(\langle m, l \rangle, x) = \langle Ack(x, m), l \rangle$
$\mathcal{J}_{\mathtt{mkdiv}}(a, x) = a$	$\mathcal{J}_{\mathtt{mkdiv}}(\langle m, l \rangle, x) = \langle m, l \rangle$

The interpretations for id, twice, min, quot, sma and ack are unchanged as list does not occur in their type. We can orient the len rules using $\mathcal{J}_{\mathtt{len}}(\langle m, l \rangle) = l$.

Continuing our example, we orient $\mathcal{R}_{\mathtt{fold}}$ with $\mathcal{J}_{\mathtt{fold}}(F, x, \langle m, l \rangle) = (d \mapsto F(d, m))^l(x)$, so using repeated function application. To see that this works, denote $[\![a]\!] = \langle m, l \rangle$. Then:

$$\begin{array}{rcl} [\![\mathtt{fold}\ F\ x\ (\mathtt{cons}\ y\ a)]\!] & = & (d \mapsto F(d, \max(y, m)))^{l+1}(x) \\ & = & (d \mapsto F(d, \max(y, m)))^l((d \mapsto F(d, \max(y, m)))(x)) \\ & = & (d \mapsto F(d, \max(y, m)))^l(F(x, \max(y, m))) \\ & \geq & (d \mapsto F(d, m))^l(F(x, y)) \text{ by weak monotonicity of } F \\ & = & [\![\mathtt{fold}\ F\ (F\ x\ y)\ a]\!] \end{array}$$

Non-numeric interpretations. As observed before, we cannot orient the `inc` rule if $[\![\mathtt{s}\ x]\!] > [\![x]\!]$, which is currently the case. To handle this problem, we must backtrack again, and update $\mathcal{A}_{\mathsf{nat}}$. Let $X = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ with $\mathtt{a} > \mathtt{b}$ and $\mathtt{a} > \mathtt{c}$. We let $\mathcal{A}_{\mathsf{nat}} = \mathbb{N} \times X$, and set:

$$\begin{array}{rclrcl} \mathcal{J}_{0} & = & \langle 0, \mathtt{b}\rangle & \mathcal{J}_{\mathtt{s}}(\langle n, e\rangle) & = & \langle n+1, \mathtt{c}\rangle \\ \mathcal{J}_{\mathtt{nil}} & = & \langle 0, 0\rangle & \mathcal{J}_{\mathtt{cons}}(\langle n, e\rangle, \langle m, l\rangle) & = & \langle \max(n, m), l+1\rangle \end{array}$$

(Note that we had to adapt $\mathcal{J}_{\mathsf{cons}}$ because it takes a `nat` as argument, but the interpretation is essentially unchanged: the new component is simply discarded.)

With this interpretation, $[\![\mathtt{s}\ 0]\!] = \langle 1, \mathtt{c}\rangle \not\sqsupset_{\mathsf{nat}} \langle 0, \mathtt{b}\rangle = [\![0]\!]$. Now we can orient the `inc` rule using: $\mathcal{J}_{\mathtt{inc}}(x, e) =$ "if $e = \mathtt{c}$ then 0 else 1". Then $[\![\mathtt{inc}\ 0]\!] = 1 = \mathtt{s}\ (\mathtt{inc}\ (\mathtt{s}\ 0))$. We update the existing interpretations by replacing references to a natural number x by its first component, and letting the second component of every defined symbol be `a`:

$$\begin{array}{rclrcl} \mathcal{J}_{\mathtt{id}}(\langle n, e\rangle & = & \langle n, \mathtt{a}\rangle & \mathcal{J}_{\mathtt{twice}}(F, \langle n, e\rangle) & = & F\ (F\ \langle n, e\rangle) \\ \mathcal{J}_{\mathtt{min}}(\langle n, e\rangle, \langle m, i\rangle) & = & \langle n, \mathtt{a}\rangle & \mathcal{J}_{\mathtt{ack}}(\langle n, e\rangle) & = & \langle Ack(n), \mathtt{a}\rangle \\ \mathcal{J}_{\mathtt{quot}}(\langle n, e\rangle) & = & \langle n, \mathtt{a}\rangle & \mathcal{J}_{\mathtt{map}}(F, \langle m, l\rangle) & = & \langle F(\langle m, \mathtt{a}\rangle), l\rangle \\ \mathcal{J}_{\mathtt{sma}}(b, F, \langle n, e\rangle) & = & \langle n, \mathtt{a}\rangle & \mathcal{J}_{\mathtt{mkbig}}(\langle m, l\rangle, \langle n, e\rangle) & = & \langle Ack(n, m), l\rangle \\ \mathcal{J}_{\mathtt{hd}}(\langle m, l\rangle) & = & \langle m, \mathtt{a}\rangle & \mathcal{J}_{\mathtt{mkdiv}}(\langle m, l\rangle, \langle n, e\rangle) & = & \langle m, l\rangle \\ \mathcal{J}_{\mathtt{len}}(\langle m, l\rangle) & = & \langle l, \mathtt{a}\rangle & \mathcal{J}_{\mathtt{fold}}(F, \langle n, e\rangle, \langle m, l\rangle) & = & (d \mapsto F(d, \langle m, \mathtt{a}\rangle))^{l}(\langle n, e\rangle) \end{array}$$

Mutually recursive symbols. To handle the mutually recursive symbols `double` and `exp`, we can either find assignments for $\mathcal{J}_{\mathtt{exp}}$ and $\mathcal{J}_{\mathtt{double}}$ at the same time, or use a trick: the system is essentially unchanged if we replace these rules by the following:

$$\begin{array}{rclrcl} \mathtt{exp}\ 0\ y & \to & y & \mathtt{exp}\ (\mathtt{s}\ x)\ y & \to & \mathtt{double}\ x\ y\ 0\ \mathtt{exp} \\ \mathtt{double}\ x\ 0\ z\ F & \to & F\ x\ z & \mathtt{double}\ x\ (\mathtt{s}\ y)\ z\ F & \to & \mathtt{double}\ x\ y\ (\mathtt{s}\ (\mathtt{s}\ z))\ F \end{array}$$

Now `double` and `exp` are no longer mutually recursive, and can be handled separately. For `double`, we can choose $\mathcal{J}_{\mathtt{double}}(x, \langle y, u\rangle, \langle z, e\rangle, F) := F(x, \langle z + 2 * y, \mathtt{a}\rangle)$. Using this, the requirements for `exp` evaluate to $\mathcal{J}_{\mathtt{exp}}(\langle 0, \mathtt{b}\rangle, y) \sqsupseteq_{\mathsf{nat}} y$ and $\mathcal{J}_{\mathtt{exp}}(\langle x+1, \mathtt{c}\rangle, \langle y, e\rangle) \sqsupseteq_{\mathsf{nat}} \mathcal{J}_{\mathtt{exp}}(\langle x, u\rangle, \langle 2 * y, \mathtt{a}\rangle)$. This is satisfied with $\mathcal{J}_{\mathtt{exp}}(\langle x, u\rangle, \langle y, e\rangle) = \langle 2^{x} * y, \mathtt{a}\rangle$. Now we can find an interpretation for the *original* definition of `double` by replacing F by $\mathcal{J}_{\mathtt{exp}}$; this gives $\mathcal{J}_{\mathtt{double}}(\langle x, i\rangle, \langle y, u\rangle, \langle z, e\rangle) = \langle 2^{x} * (z + 2 * y), \mathtt{a}\rangle$.

In this case, we only had two mutually recursive symbols, so the separation was perhaps unnecessary. However, to handle a large group of mutually recursive rules, this idea may be indispensible to split it into manageable chunks. Note also that we used the higher-order capabilities of interpretations, even though the `exp` and `double` rules are first-order.

Finishing up. The last rule, $\mathtt{H}\ (\mathtt{s}\ x) \to \mathtt{H}\ (\mathtt{twice}\ \mathtt{id}\ x)$, can be handled by choosing $\mathcal{J}_{\mathtt{H}}(x) = 0$. Now, having $[\![\ell]\!] \sqsupseteq [\![r]\!]$ for all rules, we move on to step 5 of the procedure. We let $\mathcal{A}_{\mathsf{dp}} = \mathbb{N}$ and orient the DP by choosing $\mathcal{J}_{\mathtt{H}^\sharp}(\langle x, e\rangle = x$. Then, using p_1 to denote the first element of a pair p, we have $[\![\mathtt{H}\ (\mathtt{s}\ x)]\!] = [\![x]\!]_1 + 1 > [\![x]\!]_1 = \mathcal{J}_{\mathtt{id}}(\mathcal{J}_{\mathtt{id}}(x))_1 = [\![\mathtt{H}\ (\mathtt{twice}\ \mathtt{id}\ x)]\!]$ as required. Hence, the termination proof of the extended system is complete.

It is worth noting that there are many similarities between dependency pairs and this incremental procedure for interpretations. Dividing the function symbols in groups based on mutual dependencies also happens in the splitting lemma, and handling them in order so that the dependencies for a rule $\mathtt{f}\ \vec{\ell} \to r$ have been computed before $\mathcal{J}_{\mathtt{f}}$ is reminiscent of usable rules. Non-numeric interpretations like $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ can take the same role as reachability analysis in the splitting lemma. Also, *strongly* monotonic tuple interpretations (used without dependency pairs) avoid the problem that $\mathtt{f}\ \vec{x} \succeq x_i$ of Example 5, and can handle $\mathcal{R}_{\mathtt{quot}} \cup \mathcal{R}_{\mathtt{min}}$.[17]. Hence, tuple interpretations transpose DP-like reasoning to the level of rules rather than dependency pairs. In future work it might be possible to define a similar reasoning approach as the DP framework, but based on interpretations rather than dependency pairs. This may offer a powerful tool for complexity analysis similar to the DP framework for termination.

Formalisation and implementation

The procedure above illustrates how a human can find tuple interpretations in a systematic way. However, to be practically usable for systems with thousands of rules, the approach needs to be automated – and to achieve that, there is a lot of work still to be done.

- The methods to find individual interpretations should be automated. This could be done using an encoding to SAT or SMT [7, 8, 10, 24], but the existing techniques will have to be extended to for instance support repeated function application $F^n(x)$.
- The use of interpretations to sets like $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, which we used as a kind of reachability check, should be formalised and explored more deeply. The same holds for defining functions like *Ack* based on a given terminating and confluent subset of $\mathcal{R}$.
- The process to adapt existing interpretations when $\mathcal{A}_\iota$ is expanded should be formalised. To be precise, we would like to find a systematic way to modify an interpretation function $\mathcal{J}$ so that previously proven inequalities $[\![\ell]\!] \sqsupseteq [\![r]\!]$ are preserved either directly if $\ell :: \kappa \neq \iota$, or in the first component (i.e., $[\![\ell]\!]_1 \sqsupseteq_\iota [\![r]\!]_1$) if $\ell :: \iota$. This was straightforward in all examples that we have seen, but it is not easy to define an algorithm. We *conjecture* that this can be done in general, but it may require also changing $\mathcal{A}_\kappa$ for some other sorts. If the conjecture is false, we could alternatively do a true backtracking step, and recompute all interpretations. Doing this means repeatedly discarding prior work, but it has the advantage that, with the new information, we may be able to find tighter interpretations. (For example, with $[\![\mathtt{nat}]\!] = \mathbb{N} \times \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, there is a smaller choice for $\mathcal{J}_{\mathtt{min}}$.)
- When splitting a group of mutually recursive symbols, the choice of *which* function symbol to give an extra argument to matters. In the example, replacing the exp rules by $\mathtt{exp}\ \mathtt{0}\ y\ F \to y$ and $\mathtt{exp}\ (\mathtt{s}\ x)\ y\ F \to F\ x\ y\ \mathtt{0}$ would not have given the same good result, since there is no perfectly tight interpretation for these rules. Hence, we should either find a good heuristic to choose the symbol, or use a procedure based on trial and error.

6 Conclusions

In this paper, we explored a group of methods that can be combined to build termination proofs for many large higher-order TRSs, in an incremental way. The foundation is the *static DP approach*, with techniques lifted from the first-order setting but adapted to higher-order rewriting: the splitting lemma, two subterm criteria and two usable rules lemmas. As a reduction pair, we considered weakly monotonic interpretations to *tuples*, an idea originating in complexity analysis which avoids many limitations of interpretations to $\mathbb{N}$. Most of the theory is not new (though it is adapted to a different formalism), but is used in a new way, to hopefully provide insights on the challenge of large higher-order termination problems.

A part of the techniques discussed in this paper have been implemented in Wanda [15], but not yet usable rules with respect to an argument filtering, or any form of tuple interpretations. An obvious goal for future work is to complete this implementation, and to formalise and implement the ideas of Section 5. In addition, an important goal is to transpose the methodology (and implementation) to functional programming languages. This would also allow us to investigate the power of the framework on real systems. While the termination problem database [22] does contain large systems, these are invariably first-order systems with only a few, mostly very simple, higher-order rules.

Finally, there are many ways to improve the DP framework. This could take the form of lifting more ideas from the first-order setting, recognising more situations where not all rules need to be usable (such as the DP for the H rule), or finding a way to weaken or drop the AFP restriction, for instance by combining static and dynamic dependency pairs.

References

1 T. Aoto and Y. Yamada. Dependency pairs for simply typed term rewriting. In *Proc. RTA '05*, volume 3467 of *LNCS*, pages 120–134, 2005.

2 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

3 F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Proc. RTA '00*, volume 1833 of *LNCS*, pages 47–61, 2000.

4 F. Blanqui, J. Jouannaud, and M. Okada. Inductive-data-type systems. *Theoretical Computer Science*, 272(1-2):41–68, 2002.

5 F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering: The end of a quest. In *Proc. CSL '08*, volume 5213 of *LNCS*, pages 1–14, 2008.

6 F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering. *Logical Methods in Computer Science*, 11(4), 2015.

7 C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. SAT modulo linear arithmetic for solving polynomial constraints. *Journal of Automated Reasoning*, 48(1):107–131, 2012.

8 C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. SAT '07*, volume 4501 of *LNCS*, pages 340–354. Springer, 2007.

9 C. Fuhs and C. Kop. Harnessing first order termination provers using higher order dependency pairs. In *Proc. FroCoS '11*, volume 6989 of *LNAI*, pages 147–162, 2011.

10 C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. In *Proc. RTA '12*, volume 15 of *LIPIcs*, pages 176–192, 2012.

11 C. Fuhs and C. Kop. A static higher-order dependency pair framework. In *Proc. ESOP '19*, volume 11423 of *LNCS*, pages 752–782, 2019.

12 J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, volume 3452 of *LNAI*, pages 301–331, 2005.

13 M. Hamana. Modular termination for second-order computation rules and application to algebraic effect handlers. Arxiv preprint arXiv:1912.03434, 2019.

14 J. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proc. LICS '99*, IEEE, pages 402–411, 1999.

15 C. Kop. WANDA – a higher-order termination tool. In *Proc. FSCD 20*, volume 167 of *LIPIcs*, pages 36:1–36:19, 2020.

16 C. Kop and F. van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *Logical Methods in Computer Science*, 8(2):10:1–10:51, 2012.

17 C. Kop and D. Vale. Tuple interpretations for higher-order complexity. In *Proc. FSCD '21*, volume 195 of *LIPIcs*, pages 31:1–31:22. Dagstuhl, 2021.

18 K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Transactions on Information and Systems*, 92(10):2007–2015, 2009.

19 K. Kusakari and M. Sakai. Enhancing dependency pair method using strong computability in simply-typed term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 18(5):407–431, 2007.

20 J. van de Pol. Termination proofs for higher-order rewrite systems. In *Proc. HOA 94*, volume 816 of *LNCS*, pages 305–325, 1994.

21 Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141–143, 1987.

22 Wiki. Termination Problems DataBase (TPDB). `http://termination-portal.org/wiki/TPDB`.

23 Wiki. The International Termination Competition (TermComp). `http://termination-portal.org/wiki/Termination_Competition`, 2018.

24 A. Yamada. Multi-dimensional interpretations for termination of term rewriting. In *Proc. CADE 21*, volume 12699 of *LNAI*, pages 273–290, 2021.