A Combination Framework for Complexity[☆]

Martin Avanzini[a], Georg Moser[a]

[a] *Institute of Computer Science, University of Innsbruck, Austria*

Abstract

In this paper we present a combination framework for the automated polynomial complexity analysis of term rewrite systems. The framework covers both *derivational* and *runtime complexity* analysis, and is employed as theoretical foundation in the automated complexity tool TCT. We present generalisations of powerful complexity techniques, notably a generalisation of *complexity pairs* and *(weak) dependency pairs*. Finally, we also present a novel technique, called *dependency graph decomposition*, that in the dependency pair setting greatly increases modularity.

Keywords: Term Rewriting, Resource Analysis, Runtime Complexity, Automation
2000 MSC: 68Q42,
2000 MSC: 03D15,
2000 MSC: 03B70

1. Introduction

In *implicit computational complexity* (ICC for short) one often studies abstractions of real programming languages in order to more clearly study the computational principle if only bounded resources are available [1, 2]. One abstract programming language framework often studied are *term rewrite systems* (TRSs for short) and not surprisingly methods developed in ICC are applicable to measure the complexity of rewrite systems [3].

In order to measure the complexity of a TRS it is natural to look at the maximal length of derivation sequences—the *derivation length*—as suggested by Hofbauer and Lautemann in [4]. The resulting notion of complexity is called *derivational complexity*. Based on earlier notions by Bonfante et al., Hirokawa and the second author introduced in [5] a variation, called *runtime complexity*, that only takes *basic* or *constructor-based* terms as start terms into account. The restriction to basic terms allows one to accurately express the complexity of a program through the runtime complexity of a TRS. Noteworthy both notions constitute an *invariant cost model* for rewrite systems [6, 7]. Thus techniques developed for complexity analysis of rewrite systems become readily applicable in the implicit characterisation of complexity classes, a fact well documented in the literature.

The body of research in the field of complexity analysis of rewrite systems provides a wide range of different techniques to analyse the time complexity of rewrite systems, fully automatically. Techniques range from *direct methods*, like *polynomial path orders* [8, 9] and other suitable restrictions of termination orders [3, 10], to *transformation techniques*, maybe most prominently adaptations of the *dependency pair method* [5, 11], *semantic labeling* over finite carriers [12], methods to combine base techniques [13] and the *weight gap principle* [5, 13]. (See [14] for an overview of complexity analysis methods for term rewrite systems.) In particular the dependency pair method for complexity analysis allows for a wealth of techniques originally intended for termination analysis. We mention *(safe) reduction pairs* [5], *various rule transformations* [11],

[☆] This work is partly supported by FWF (Austrian Science Fund) project I-608-N18.
Email addresses: martin.avanzini@uibk.ac.at (Martin Avanzini), georg.moser@uibk.ac.at (Georg Moser)

and *usable rules* [5]. Some very effective methods have been introduced specifically for complexity analysis in the context of dependency pairs. For instance, *path analysis* [5, 15, 16] decomposes the analysed rewrite relation into simpler ones, by treating paths through the *dependency graph* independently. *Knowledge propagation* [11] is another complexity technique relying on dependency graph analysis, which allows one to propagate bounds for specific rules along the dependency graph. Besides these, various minor simplifications are implemented in tools, mostly relying on dependency graph analysis. With this paper, we provide following contributions.

1. We propose a uniform *combination framework for complexity analysis*, that is capable of expressing the majority of the rewriting based complexity techniques in a unified way. Such a framework is essential for the development of a modern complexity analyser for term rewrite systems. The implementation of our complexity analyser TCT [17], the *Tyrolean Complexity Tool*, closely follows the formalisation proposed in this work. Noteworthy, TCT is currently the only tool that participates in all four complexity sub-divisions of the annual *termination competition*.[1]

2. A majority of the cited techniques were introduced in restricted or incompatible contexts. For instance, in [13] the derivational complexity of relative TRSs is considered. Conversely, neither [5, 16] nor [11] treat relative systems, and restrict their attention to basic start terms. Where non-obvious, we generalise these techniques to our setting. Noteworthy, our notion of $\mathcal{P}$*-monotone complexity pair* generalises complexity pairs from [13] for derivational complexity, μ*-monotone complexity pairs* for runtime complexity analysis [16, 18], and *safe reduction pairs* studied in [5, 11] that work on dependency pairs.[2] We also generalise the two different forms of dependency pairs for complexity analysis introduced in [5] and [11]. This for instance allows our tool TCT to employ these powerful techniques on a TRS $\mathcal{R}$ relative to some theory expressed as a TRS $\mathcal{S}$.

3. We introduce a novel proof technique for runtime complexity analysis called *dependency graph decomposition*. Resulting sub-problems are syntactically of a simpler form, and the analysis of these sub-problems is often easier. Importantly, the sub-problems are usually also computationally simpler in the sense that their complexity is strictly smaller than the one of the input problem. If the complexity of the two generated sub-problems is bounded by a function in $\mathcal{O}(f)$ and $\mathcal{O}(g)$ respectively, then the complexity of the input is bounded by $\mathcal{O}(f \cdot g)$. Experiments conducted with TCT indicate that this estimation is often asymptotically precise.

As for the motivation of the results established here we want to emphasise the fact that in (static) program analysis *modularity* (aka *composability*) of the techniques is of utmost importance and often considered crucial (we exemplarily mention [19–21]). Similarly the concept of *modularity* is present in ICC literature, cf. [22]. This is in stark contrast to existing results in the literature on complexity of rewriting. Our framework, in particular the dependency graph decomposition method, overcome this deficiency to a certain degree. A fact that is also observable through the provided experimental data.

Partly the results established here have already been presented in the conference paper [23]. In contrast to the conference version we here provide full proofs and have striven for an elaborate description of the adaptation of existing techniques within the novel complexity framework. Furthermore we carefully crafted suitable examples showing the intrinsic expressivity of the proposed framework. Some parts of this article are part of the first authors PhD thesis [24].

Related Work. Polynomial complexity analysis is an active research area in rewriting. While the concept has received some attention quite early by work of Choppy et al. [25], only recently the field matured. As mentioned above to a great extend these techniques are influenced, if not based on, principles stemming from the *implicit computational complexity* area. For instance [8, 9] provides a syntactic complexity analysis technique which essentially embodies *tiered recursion* in the form proposed by Bellantoni and Cook [26]. The

[1] `http://www.termination-portal.org/wiki/Termination_Competition/`.
[2] In [11] safe reductions pairs are called COM-*monotone reduction pairs*.

orders we discuss in Section 4.1 are closely related to the work of Bonfante et al. [3], where a characterisation of the polytime computable functions is proposed.

We also want to mention ongoing approaches for the automated analysis of resource usage in programs. Notably, Hoffmann et al. [27] provide an automatic multivariate amortised cost analysis exploiting typing, which extends earlier results on amortised cost analysis. Finally Albert et al. [28] present an automated complexity tool for Java™ Bytecode programs, Alias et al. [29] give a complexity and termination analysis for flowchart programs, and Gulwani and Zuleger [19] as well as Zuleger et al. [20] provide an automated complexity tool for C programs. Very recently Hofmann and Rodriguez proposed in [30] an automated resource analysis for object-oriented programs via an amortised cost analysis. In all this works, composability is a key issue.

Outline. The remainder of this paper is organised as follows. In the next section we cover some basics. Our *combination framework* is then introduced in Section 3. In Section 4 we show how various existing techniques can be suitable generalised for integration into our framework. Furthermore in this section we also introduce the novel method of *dependency graph decomposition*. In Section 5 we shortly report on our implementation and provide experimental evidence of the viability of our method. Finally, we conclude in Section 6.

2. Preliminaries

We denote by $\mathbb{N}$ the set of natural numbers. Let R be a binary relation. The transitive closure of R is denoted by R^+ and its transitive and reflexive closure by R^*. For $n \in \mathbb{N}$ we denote by R^n the *n-fold composition of R*. The binary relation R is *well-founded* (on a set A) if there exists no infinite chain $a_0, a_1, \ldots$ with $a_i \ R \ a_{i+1}$ for all $i \in \mathbb{N}$ ($a_0 \in A$). The relation R is *finitely branching* if for all elements a, the set $\{b \mid a \ R \ b\}$ is finite. A *pre-order* is a reflexive and transitive binary relation.

To compare partial functions we use *Kleene equality*: two partial functions $f, g : A \to B$ are equal, in notation $f =_{\mathsf{k}} g$, if for all $a \in A$ either $f(a)$ and $g(a)$ are defined and $f(a) = g(a)$, or both $f(a)$ and $g(a)$ are undefined. We write $f \geqslant_{\mathsf{k}} g$ if for all $a \in A$ with $g(a)$ defined, $f(a)$ is defined and $f(a) = g(a)$ holds. Then $f =_{\mathsf{k}} g$ if and only if $f \geqslant_{\mathsf{k}} g$ and $g \geqslant_{\mathsf{k}} f$.

Term Rewriting. We assume basic familiarity with rewriting [31] and only fix notations here. We denote by $\mathcal{V}$ a countably infinite set of *variables* and by $\mathcal{F}$ a *signature*, i.e. a finite set of function symbols. Each function symbol $f \in \mathcal{F}$ is equipped with a natural number k, the *arity* of f, which we also indicate by writing $f/k \in \mathcal{F}$. The signature $\mathcal{F}$ and variables $\mathcal{V}$ are fixed throughout the paper. The set of terms $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is the least set that contains all variables, and that contains $f(t_1, \ldots, t_k)$ whenever $t_1, \ldots, t_k \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and $f/k \in \mathcal{F}$. The *root*, i.e. topmost, symbol of a term t is denoted by $\mathsf{rt}(t)$. A term is *ground* if it contains no variables. We suppose a partitioning of $\mathcal{F}$ into *constructors* $\mathcal{C}$ and *defined symbols* $\mathcal{D}$. Ground terms over $\mathcal{C}$ are called *values*. A term t is called *basic* if its root symbol is a defined symbol and all arguments are values. The set of basic terms is denoted by $\mathcal{T}_{\mathsf{b}}(\mathcal{D}, \mathcal{C})$. Terms are denoted by $s, t, u, \ldots$, possibly followed by subscripts. We denote by $|t|$ the *size* of t, i.e., the number of occurrences of symbols in t.

A *position* is a finite sequence of positive natural numbers. We denote by ϵ the *empty position*, and $p{\cdot}q$ denotes the *concatenation* of positions p and q. We say that a position p is *above* a position q if there exists a position r such that $p{\cdot}r = q$. If p is above q we also say that q is *below* r, and we write $p \leqslant q$. We write $p < q$ if $p \leqslant q$ and $p \neq q$. Positions p and q are called *parallel*, in notation $p \parallel q$, if neither $p \leqslant q$ nor $q \leqslant r$ holds. We use $t|_p$ to refer to the *subterm* of the term t at position p, recursively defined as follows: $t|_\epsilon := t$ and $t|_{i \cdot p} := t_i|_p$ where $t = f(t_1, \ldots, t_n)$ and $i \in \{1, \ldots, n\}$. The set of all positions in t, i.e. the set of positions p where $t|_p$ is defined, is denoted by $\mathcal{P}\mathrm{os}(t)$. For a set of symbols $\mathcal{X}$, we use $\mathcal{P}\mathrm{os}_{\mathcal{X}}(t) \subseteq \mathcal{P}\mathrm{os}(t)$ to denote the set of all positions p such that the root of the sub-term $r|_p$ is a symbol from $\mathcal{X}$. For $\mathcal{X} = \{x\}$ a singleton we also write $\mathcal{P}\mathrm{os}_x(t)$ instead of $\mathcal{P}\mathrm{os}_{\mathcal{X}}(t)$.

Let $\Box \notin \mathcal{F}$ denote a fresh constant, the *hole*. Elements $C \in \mathcal{T}(\mathcal{F} \cup \{\Box\}, \mathcal{V})$ with n occurrences of $\Box$ are called *n-holed* contexts. For an n-holed context C and terms $t_1, \ldots, t_n$, we denote by $C[t_1, \ldots, t_n]$ the term obtained by replacing the n holes in C by the terms $t_1, \ldots, t_n$ from left to right. A substitution is a finite

mapping $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ from variables to terms, substitutions are denoted by $\sigma, \tau, \ldots$. We denote by σ also the homomorphic extension of σ from terms to terms, and write $t\sigma$ instead of $\sigma(t)$.

Let $\to$ be a binary relation on terms. The relation $\to$ is called *stable under substitutions* if $s \to t$ implies $s\sigma \to t\sigma$ for every substitution σ. It is *closed under contexts* if $s \to t$ implies $C[s] \to C[t]$ for all contexts C. The relation $\to$ is a *rewrite relation* if it is closed under contexts and stable under substitutions. For a set of terms $T \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V})$, we define $\to(T) := \{t \mid \exists s \in T.\ s \to t\}$.

A *rewrite rule* is a pair (l, r) of terms, in notation $l \to r$, such that the *left-hand* side $l = f(l_1, \ldots, l_n)$ is not a variable, the *root* f is a *defined symbol*, and all variables appearing in the *right-hand* r occur also in l. A *term rewrite system* (*TRS* for short) $\mathcal{R}$ (over the signature $\mathcal{F}$ and variables $\mathcal{V}$) is a finite set of rewrite rules. We always denote by $\mathcal{R}, \mathcal{S}, \mathcal{Q}$ rewrite systems. A function symbol $f \in \mathcal{D}$ is *defined by* $\mathcal{R}$ if it is the root of a left-hand side in $\mathcal{R}$. We denote by $\to_{\mathcal{R}}$ the least rewrite relation that extends $\mathcal{R}$. The relation $\to_{\mathcal{R}}$ is called the *rewrite relation* of $\mathcal{R}$. A term t is called a *normal form* of $\mathcal{R}$ if there exists no term s with $s \to_{\mathcal{R}} t$. The set of all normal forms of $\mathcal{R}$ is denoted by $\mathrm{NF}(\mathcal{R})$. Abusing notation we extend this notion to binary relations $\to$ on terms in the obvious way.

For two TRSs $\mathcal{Q}$ and $\mathcal{R}$, we define $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t$ if there exist a context C, substitution σ, and rule $f(l_1, \ldots, l_k) \to r \in \mathcal{R}$ such that $s = C[f(l_1\sigma, \ldots, l_k\sigma)]$, $t = C[r\sigma]$ and all terms $l_i\sigma$ $(i = 1, \ldots, k)$ are $\mathcal{Q}$ normal forms. The subterm $f(l_1\sigma, \ldots, l_k\sigma)$ of s is called the *redex* of the step $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t$. A (possible infinite) sequence of steps $t_0 \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t_1 \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t_2 \xrightarrow{\mathcal{Q}}_{\mathcal{R}} \cdots$ is called a $\xrightarrow{\mathcal{Q}}_{\mathcal{R}}$ *rewrite sequence*, or *derivation*. For clarity, we sometimes underline the redex when we depict steps $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t$. Note that by definition $\to_{\mathcal{R}} = \xrightarrow{\varnothing}_{\mathcal{R}}$, and that the relation $\xrightarrow{\mathcal{R}}_{\mathcal{R}}$ corresponds to the *innermost rewrite relation*, where arguments have to be reduced first. We extend $\xrightarrow{\mathcal{Q}}_{\mathcal{R}}$ to a relative setting and define for TRSs $\mathcal{R}$ and $\mathcal{S}$ the relation $\xrightarrow{\mathcal{Q}}_{\mathcal{R}/\mathcal{S}} := \xrightarrow{\mathcal{Q}}{}^*_{\mathcal{S}} \cdot \xrightarrow{\mathcal{Q}}_{\mathcal{R}} \cdot \xrightarrow{\mathcal{Q}}{}^*_{\mathcal{S}}$, which is called the *$\mathcal{Q}$-restricted rewrite relation of $\mathcal{R}$ modulo $\mathcal{S}$*. We emphasise that $\mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{R}/\mathcal{S}})$, when defined, gives an upper bound on the number of $\mathcal{R}$-steps in $\xrightarrow{\mathcal{Q}}_{\mathcal{R}\cup\mathcal{S}}$ rewrite sequences starting from t.

The *derivation height* of a term t with respect to the binary relation $\to$ is given by $\mathsf{dh}(t, \to) := \max\{n \mid \exists t_0, \ldots, t_n.\ t = t_0 \to t_1 \to \cdots \to t_n\}$. Observe that $\mathsf{dh}(t, \to)$ is defined whenever $\to$ is finitely branching and well-founded. For a set of *starting terms* T and $n \in \mathbb{N}$ we define

$$\mathsf{cp}(n, T, \to) := \max\{\mathsf{dh}(t, \to) \mid \exists t \in T.\ |t| \leqslant n\}\ .$$

The *derivational complexity* of a TRS $\mathcal{R}$ is given by $\mathrm{dc}_{\mathcal{R}}(n) := \mathsf{cp}(n, \mathcal{T}(\mathcal{F}, \mathcal{V}), \to_{\mathcal{R}})$ for all $n \in \mathbb{N}$. The *runtime complexity* takes only basic terms as starting terms into account: $\mathrm{rc}_{\mathcal{R}}(n) := \mathsf{cp}(n, \mathcal{T}_{\mathsf{b}}(\mathcal{D}, \mathcal{C}), \to_{\mathcal{R}})$ for all $n \in \mathbb{N}$. By exchanging $\to_{\mathcal{R}}$ with $\xrightarrow{\mathcal{R}}_{\mathcal{R}}$ we obtain the notions of *innermost derivational* and *innermost runtime complexity*, respectively.

3. A Combination Framework for Complexity Analysis

In this section we introduce the *complexity framework* underlying TCT. The proposed framework is influenced to a great extent by the work of Thiemann [32] on the dependency pair framework for termination analysis. The notion of *complexity processor*, or simply *processor*, lies at the heart of our framework. A complexity processor dictates how to transform the analysed input *problem* into (hopefully) simpler sub-problems. It also relates the complexity of the obtained sub-problems to the complexity of the input problem. In our framework, such a processor is modeled as a set of inference rules

$$\frac{\vdash \mathcal{P}_1 \colon f_1 \quad \cdots \quad \vdash \mathcal{P}_n \colon f_n}{\vdash \mathcal{P} \colon f}\ ,$$

over judgements of the form $\vdash \mathcal{P} \colon f$. Here $\mathcal{P}$ denotes a *complexity problem* (*problem* for short) and $f : \mathbb{N} \to \mathbb{N}$ a *bounding function*. The validity of a judgement $\vdash \mathcal{P} \colon f$ is given when the function f binds the complexity of the problem $\mathcal{P}$ asymptotically.

Conceptually, a complexity problem $\mathcal{P}$ consists of a set of *starting terms* $\mathcal{T}$ together with a relation $\to_{\mathcal{S}\cup\mathcal{W}}$ for TRSs $\mathcal{S}, \mathcal{W}$. The complexity function $\mathsf{cp}_{\mathcal{P}} : \mathbb{N} \to \mathbb{N}$ of $\mathcal{P}$ accounts for the number of applications of rules from $\mathcal{S}$ in derivations starting from terms $t \in \mathcal{T}$, measured in the size of t. The component $\mathcal{W}$ of $\mathcal{P}$

can be used to express rewriting of $\mathcal{S}$ relative to a theory. It will also be used in subsequently introduced processors.

Definition 1 (Complexity Problem, Complexity Function)**.**

1. A *complexity problem* $\mathcal{P}$ (*problem* for short) is a quadruple $(\mathcal{S}, \mathcal{W}, \mathcal{Q}, \mathcal{T})$, in notation $\langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$, where $\mathcal{S}, \mathcal{W}, \mathcal{Q}$ are TRSs and $\mathcal{T} \subseteq \mathcal{T}(\mathcal{F})$ a set of ground terms. We call $\mathcal{S}$ and $\mathcal{W}$ the *strict* and *weak component* of $\mathcal{P}$ respectively. The set $\mathcal{T}$ is called the set of *starting terms* of $\mathcal{P}$.
2. The *rewrite relation of* $\mathcal{P}$ is defined as $\to_{\mathcal{P}} := \xrightarrow{\mathcal{Q}}_{\mathcal{S} \cup \mathcal{W}}$, a derivation $t \to_{\mathcal{P}} t_1 \to_{\mathcal{P}} \cdots$ is also called a $\mathcal{P}$*-derivation* (*starting from* t).
3. The *complexity (function)* $\mathsf{cp}_{\mathcal{P}} : \mathbb{N} \to \mathbb{N}$ of $\mathcal{P}$ is defined as the partial function
$$\mathsf{cp}_{\mathcal{P}}(n) := \mathsf{cp}(n, \mathcal{T}, \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}}) \,.$$

In the sequel $\mathcal{P}$, always denotes a complexity problem. We always use $\mathcal{S}$ and $\mathcal{W}$ for the strict and weak component of a complexity problem, whereas $\mathcal{R}$ refers to a set of rewrite rules that can occur in both components. These notations are possibly followed by subscripts. We write $l \to r \in \mathcal{P}$ for $l \to r \in \mathcal{S} \cup \mathcal{W}$, where $\mathcal{S}$ and $\mathcal{W}$ are the strict and weak component of $\mathcal{P}$ respectively.

Consider a problem $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$. If $\xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}}$ is *terminating* and *finitely branching* on $\mathcal{T}$, then the complexity function $\mathsf{cp}_{\mathcal{P}}$ is defined on all inputs, by König's Lemma. The following example shows that in the relative setting the termination property alone does not suffice that $\mathsf{cp}_{\mathcal{P}}$ is defined on all inputs.

Example 1. Consider the problem $\mathcal{P}_1 := \langle \mathcal{S}_1/\mathcal{W}_1, \varnothing, \mathcal{T}_1 \rangle$ where $\mathcal{S}_1 := \{\mathsf{g}(\mathsf{s}(x)) \to \mathsf{g}(x)\}$, $\mathcal{W}_1 := \{\mathsf{f}(x) \to \mathsf{f}(\mathsf{s}(x)),\ \mathsf{f}(x) \to \mathsf{g}(x)\}$, and $\mathcal{T}_1 := \{\mathsf{f}(\bot)\}$. Observe that $\to_{\mathcal{P}_1}$ derivations are of the form
$$\mathsf{f}(\bot) \to^*_{\mathcal{W}_1} \mathsf{f}(\mathsf{s}^n(\bot)) \to_{\mathcal{W}_1} \mathsf{g}(\mathsf{s}^n(\bot)) \to^n_{\mathcal{S}_1} \mathsf{g}(\bot) \,,$$
for $n \in \mathbb{N}$, that is, $\mathsf{f}(\bot) \to^n_{\mathcal{S}_1/\mathcal{W}_1} \mathsf{g}(\bot)$ holds for all $n \in \mathbb{N}$. Thus, whereas $\to_{\mathcal{S}_1/\mathcal{W}_1}$ is well-founded on $\mathcal{T}_1$, $\mathsf{cp}_{\mathcal{P}_1}(m) =_{\mathsf{k}} \mathsf{dh}(\mathsf{f}(\bot), \to_{\mathcal{S}_1/\mathcal{W}_1})$ is undefined for $m \geqslant 2$.

The example exploits that the rewrite relation, although well-founded, is not finitely branching. The next example shows that for a complexity problem $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$, even if $\xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}}$ is not finitely branching on $\mathcal{T}$, the complexity function $\mathsf{cp}_{\mathcal{P}}$ can still be defined on all inputs.

Example 2 (Continued from Example 1). Consider the complexity problem $\mathcal{P}_2 := \langle \mathcal{S}_2/\mathcal{W}_1, \varnothing, \mathcal{T}_1 \rangle$, where $\mathcal{S}_2 := \{\mathsf{g}(x) \to x\}$. The complexity function of $\mathcal{P}_2$ is constant, but $\mathsf{f}(\bot) \to_{\mathcal{S}_2/\mathcal{W}_1} \mathsf{s}^n(\bot)$ holds for all $n \in \mathbb{N}$, i.e. $\to_{\mathcal{S}_2/\mathcal{W}_1}$ is not finitely branching on $\mathcal{T}_1$.

The following notions of *innermost*, *runtime* and *derivational complexity problem* allow us to carry over techniques for rewrite systems, applicable in these restricted contexts, to complexity problems.

Definition 2 (Runtime, Derivational, Innermost Complexity Problem)**.** Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$ denote a complexity problem.

1. Then $\mathcal{P}$ is called a *runtime complexity problem* if $\mathcal{T} \subseteq \mathcal{T}_{\mathsf{b}}(\mathcal{D}, \mathcal{C})$ holds. Otherwise it is called a *derivational complexity problem*.
2. The problem $\mathcal{P}$ is called an *innermost complexity problem* if $\mathrm{NF}(\mathcal{Q}) \subseteq \mathrm{NF}(\mathcal{S} \cup \mathcal{W})$.

As for runtime complexity analysis of rewrite systems, a runtime complexity problem takes only basic starting terms into account. For an innermost complexity problem $\mathcal{P}$ as above, the rewrite relation $\to_{\mathcal{P}}$ is included in the innermost rewrite relation of $\mathcal{R} \cup \mathcal{S}$.

Definition 3 (Judgement, Processor, Proof)**.**

1. A *(complexity) judgement* is a statement $\vdash \mathcal{P} \colon f$ where $\mathcal{P}$ is a complexity problem and $f : \mathbb{N} \to \mathbb{N}$. The judgement is *valid* if $\mathsf{cp}_{\mathcal{P}}$ is defined on all inputs, and $\mathsf{cp}_{\mathcal{P}} \in \mathcal{O}(f)$.

2. A *complexity processor* Proc (*processor* for short) is an inference rule

$$\frac{\vdash \mathcal{P}_1 \colon f_1 \quad \cdots \quad \vdash \mathcal{P}_n \colon f_n}{\vdash \mathcal{P} \colon f}\ \mathsf{Proc}\ ,$$

over complexity judgements. The problems $\mathcal{P}_1, \ldots, \mathcal{P}_n$ are called the *sub-problems generated by* Proc *on* $\mathcal{P}$. The processor Proc is *sound* if $\vdash \mathcal{P} \colon f$ is valid whenever the judgements $\vdash \mathcal{P}_1 \colon f_1, \ \ldots, \ \vdash \mathcal{P}_n \colon f_n$ are valid. The processor is *complete* if the inverse direction holds.

3. Let empty denote the axiom $\vdash \langle \varnothing / \mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle \colon f$ for all TRSs $\mathcal{W}$ and $\mathcal{Q}$, set of terms $\mathcal{T}$ and $f : \mathbb{N} \to \mathbb{N}$. A *complexity proof* (*proof* for short) of a judgement $\vdash \mathcal{P} \colon f$ is a deduction using sound processors from the axiom empty and *assumptions* $\vdash \mathcal{P}_1 \colon f_1, \ldots, \vdash \mathcal{P}_n \colon f_n$, in notation $\mathcal{P}_1 \colon f_1, \ldots, \mathcal{P}_n \colon f_n \vdash \mathcal{P} \colon f$.

We say that a complexity proof is *closed* if its set of assumptions is empty, otherwise it is *open*. We follow the usual convention and annotate side conditions as premises to inference rules. When the list of premises in a processor

$$\frac{\vdash \mathcal{P}_1 \colon f_1 \quad \cdots \quad \vdash \mathcal{P}_n \colon f_n}{\vdash \mathcal{P} \colon f}\ \mathsf{Proc}\ ,$$

is empty, i.e. $n = 0$, we call Proc also a *direct processor*. The notion of *complexity pair* introduced in Section 4.1 for example is an instance of a direct processor. Processors that are not direct processors are called *transformations*.

Soundness of a processor guarantees that our formal system is correct. Completeness ensures that a deduction gives asymptotically tight bounds.

Theorem 1. *If there exists a closed complexity proof* $\vdash \mathcal{P} \colon f$*, then the judgement* $\vdash \mathcal{P} \colon f$ *is valid.*

Proof. The theorem follows by a standard induction on the size of proofs, exploiting that the underlying set of processors is sound. □

As we see in the sequel, our formalisation is expressive enough to cover a majority of the techniques available for the automated complexity analysis. To justify our design choices, we briefly compare our formulations to previously established notions.

Related Complexity Frameworks. Our notion of complexity problem is a natural extension of the one established by Zankl and Korp [13] for derivational complexity analysis, which underlies the complexity tool CaT[3]. CaT is a variation of the fast and powerful complexity analyser TcT 2 [33]. In the framework underlying CaT, a complexity problem consists of a *relative rewrite system* $\mathcal{S}/\mathcal{W}$ together with a bounding function $f : \mathbb{N} \to \mathbb{N}$. A processor in this setting is a function

$$((\mathcal{S}_1 \cup \mathcal{S}_2)/\mathcal{W}, f) \mapsto (\mathcal{S}_1/(\mathcal{S}_2 \cup \mathcal{W}), f') \ ,$$

which shifts rules $\mathcal{S}_2$ from the *strict component* to the *weak component*. This processor is sound if $f(n) + \mathrm{dc}_{(\mathcal{S}_1 \cup \mathcal{S}_2)/\mathcal{W}}(n) \in \mathcal{O}(f'(n) + \mathrm{dc}_{\mathcal{S}_1/(\mathcal{S}_2 \cup \mathcal{W})}(n))$ holds, where $\mathrm{dc}_{\mathcal{S}/\mathcal{W}} := \mathsf{cp}(n, \mathcal{T}(\mathcal{F}, \mathcal{V}), \to_{\mathcal{S}/\mathcal{W}})$ extends the derivational complexity function from TRSs to relative rewrite systems. Our framework generalises these notions in various aspects. First of all, we have made the set of starting terms abstract, which allows us to cover the derivational and runtime complexity of TRSs. We also allow transformations where rules appearing in the generated sub-problem do not appear in the input problem. Instances of such transformations are for example the two notions of dependency pair transformations discussed in Section 4.3. Also, we permit processors to transform the input problem into more than one sub-problem. Finally, we do not enforce linear combinations of complexity certificates. The latter two properties are for instance necessary in the formulation of dependency graph decomposition, discussed at the end of Section 4.5.

[3] `http://cl-informatik.uibk.ac.at/software/cat/`

In the basic setting of the framework underlying AProVE [34], a complexity problem consists of a triple $(\mathcal{D}, \mathcal{S}, \mathcal{R})$ for *dependency tuples* $\mathcal{S} \subseteq \mathcal{D}$ (compare Section 4.3) and rewrite rules $\mathcal{R}$. Let $\mathcal{W} := \mathcal{D} \setminus \mathcal{S}$. The complexity of such a problem essentially amounts to the number of applications of dependency tuples from $\mathcal{S}$ in derivations $\xrightarrow{\mathcal{R}}_{\mathcal{S}/\mathcal{W}\cup\mathcal{R}}$ starting from a suitable set of terms $\mathcal{T}$. In our setting, the triple $(\mathcal{D}, \mathcal{S}, \mathcal{R})$ can thus be represented as the problem $\langle \mathcal{S}/\mathcal{W} \cup \mathcal{R}, \mathcal{R}, \mathcal{T}\rangle$. Our notion of complexity problem is more general, since we do not restrict the strict component to dependency tuples. This generality is necessary to cover full rewriting and derivational complexity analysis, because a canonical complexity problem cannot always be transformed into the restricted form.

Finally, we note that Noschinski et al. [34] also propose an extension of their notion of complexity problem, using an additional component $\mathcal{K}$ of *known* rules. These are rules whose complexity, in the sense of the number of applications of rules from $\mathcal{K}$ in the considered derivations, has already been assessed. This information can be reused in the *knowledge propagation* processor [34]. Our framework cannot capture this extension, although we could in principle extend our notion of complexity problem sufficiently. This of course incurs some complications, as in each processor the additional component $\mathcal{K}$ needs to be treated properly.

4. Complexity Processors in TCT

We now discuss various methods implemented in TCT. Some of the techniques were initially introduced in the context of termination analysis, but have been later adapted for the polynomial complexity analysis in various works. We recast these techniques to suite complexity problems. Our notion of $\mathcal{P}$*-monotone complexity pair*, covered in Section 4.1, is inspired by the notion *complexity pair* introduced by Zankl and Korp [13], but we weaken monotonicity requirements for runtime complexity analysis. $\mathcal{P}$-monotone complexity pair also generalise the notion of *safe reduction pair* proposed by Hirokawa and Moser [5]. Zankl and Korp [13] use complexity pairs in an iterated fashion, we adapt this approach in Section 4.2. In Section 4.3 we suitably adapt the two forms of *dependency pairs* proposed by Hirokawa and Moser [5] and Noschinski et al. [34]. We then generalise *usable rules* in Section 4.4 and present various simplification techniques based on a *dependency graph* analysis in Section 4.5. Here, we also propose a novel technique, called *dependency graph decomposition*. This technique constitutes a suitable adaptation of *cycle analysis* proposed by Giesl et al. [35] for the termination analysis of rewrite systems.

Throughout the rest of this section, the following two rewrite systems will serve as running examples. The first is a small toy example.

Example 3. Consider the rewrite system $\mathcal{R}_\times$ given by the following four rules.

$$1\colon 0 + y \to y \qquad 2\colon \mathsf{s}(x) + y \to \mathsf{s}(x + y) \qquad 3\colon 0 \times y \to 0 \qquad 4\colon \mathsf{s}(x) \times y \to y + (x \times y)\ .$$

This TRS computes multiplication (and addition) on numerals constructed from the constant 0 and the successor s. For $n \in \mathbb{N}$, we use the notation $\mathbf{n}$ below to abbreviate numerals $\mathsf{s}(\cdots\mathsf{s}(0)\cdots)$, where the successor s occurs n times.

Let $\mathcal{T}_\times$ denote the set of basic terms with defined symbols $+, \times$ and constructors $\mathsf{s}, 0$. We denote by $\mathcal{P}_\times$ the canonical runtime complexity problem $\langle \mathcal{R}_\times/\varnothing, \varnothing, \mathcal{T}_\times\rangle$, and by $\mathcal{P}_{\times\text{-i}}$ the *innermost* runtime complexity $\langle \mathcal{R}_\times/\varnothing, \mathcal{R}_\times, \mathcal{T}_\times\rangle$

The runtime complexity analysis of our second TRS $\mathcal{R}_\mathsf{K}$ is significantly more involved. This example implements Kruskal's algorithm for computing a spanning forest of minimal weight for a given graph, see Figure 1.

Example 4. In $\mathcal{R}_\mathsf{K}$, a graph is represented as a term $\mathsf{graph}(N, E)$ where N and E refer to the nodes and edges respectively. Sets are encoded as lists, build from the constant $[\,]$ and the binary symbol $(::)$ in the usual way. We suppose nodes and weights are given as natural numbers. For simplicity, these are encoded as tally numbers $\mathsf{s}^n(0)$. Edges e are given as triples (n, w, m), where the rules

$$5\colon \mathsf{src}((n, w, m)) \to n \qquad 6\colon \mathsf{wt}((n, w, m)) \to w \qquad 7\colon \mathsf{trg}((n, w, m)) \to m\ ,$$

Let N denote the nodes and E weighted edges of a graph G.

Set $F := \varnothing$ and $P := \{\{n\} \mid n \in N\}$.

For all edges $e \in E$, sorted increasingly by their weight, do:

  If source and target of e occur in $p, q \in P$ respectively, with $p \neq q$;
    Set $P := P \setminus \{p, q\} \cup \{p \cup q\}$ and $F := F \cup \{e\}$.

Return F.

Figure 1: Kruskal's algorithm for computing a spanning forest of minimal weight.

provide projections to the source node n, weight w, and target node m. The following rules contained in $\mathcal{R}_{\mathsf{K}}$ implement Kruskal's algorithm.

$$
\begin{aligned}
&8: && \mathsf{forest}(\mathsf{graph}(N, E)) \to \mathsf{kruskal}(\mathsf{sort}(E), [\,], \mathsf{partitions}(N))\\
&9: && \mathsf{partitions}([\,]) \to [\,]\\
&10: && \mathsf{partitions}(n :: N) \to (n :: [\,]) :: \mathsf{partitions}(N)\\
&11: && \mathsf{kruskal}([\,], W, P) \to W\\
&12: && \mathsf{kruskal}(e :: E, W, P) \to \mathsf{kruskal?}(\mathsf{inBlock}(e, P), e, E, W, P)\\
&13: && \mathsf{kruskal?}(\mathsf{tt}, e, E, W, P) \to \mathsf{kruskal}(E, W, P)\\
&14: && \mathsf{kruskal?}(\mathsf{ff}, e, E, W, P) \to \mathsf{kruskal}(E, e :: W, \mathsf{join}(e, P, [\,]))\\
&15: && \mathsf{inBlock}(e, [\,]) \to \mathsf{ff}\\
&16: && \mathsf{inBlock}(e, p :: P) \to (\mathsf{src}(e) \in p \wedge \mathsf{trg}(e) \in p) \vee \mathsf{inBlock}(e, P)\\
&17: && \mathsf{join}(e, [\,], q) \to q :: [\,]\\
&18: && \mathsf{join}(e, p :: P, q) \to \mathsf{join?}(\mathsf{src}(e) \in p \vee \mathsf{trg}(e) \in p, e, p, P, q)\\
&19: && \mathsf{join?}(\mathsf{tt}, e, p, P, q) \to \mathsf{join}(e, P, p +\!\!+ q)\\
&20: && \mathsf{join?}(\mathsf{ff}, e, p, P, q) \to p :: \mathsf{join}(e, P, q)\ .
\end{aligned}
$$

The defined symbol forest starts the computation on input graph $\mathsf{graph}(N, E)$. The rules (11)—(14) are used to iterate the loop from Figure 1. The rules (15) and (16) check the condition, and the remaining rules (17)–(20) execute the body of the conditional. To sort edges according to their weight, the TRS $\mathcal{R}_{\mathsf{K}}$ uses the following implementation of insertion sort.

$$
\begin{aligned}
&21: && \mathsf{sort}([\,]) \to [\,] && 22: && \mathsf{sort}(e :: E) \to \mathsf{insert}(e, \mathsf{sort}(E))\\
&23: && \mathsf{insert}(e, [\,]) \to e :: [\,] && 24: && \mathsf{insert}(e, f :: E) \to \mathsf{insert?}(\mathsf{wt}(e) \leqslant \mathsf{wt}(f), e, f, E)\\
&25: && \mathsf{insert?}(\mathsf{tt}, e, f, E) \to e :: (f :: E) && 26: && \mathsf{insert?}(\mathsf{ff}, e, f, E) \to f :: \mathsf{insert}(e, E)\ .
\end{aligned}
$$

On the list representation of sets, membership and union is defined as expected:

$$
\begin{aligned}
&27: && n \in [\,] \to \mathsf{ff} && 28: && n \in (m :: p) \to n = m \vee n \in p\\
&29: && [\,] +\!\!+ q \to q && 30: && (n :: p) +\!\!+ q \to n :: (p +\!\!+ q)\ .
\end{aligned}
$$

Finally, the following rules define standard Boolean operations, and comparisons on natural numbers.

$$
\begin{aligned}
&31: && 0 = 0 \to \mathsf{tt} && 32: && \mathsf{s}(x) = 0 \to \mathsf{ff} && 33: && 0 = \mathsf{s}(y) \to \mathsf{ff} && 34: && \mathsf{s}(x) = \mathsf{s}(y) \to x = y\\
&35: && 0 \leqslant 0 \to \mathsf{tt} && 36: && \mathsf{s}(x) \leqslant 0 \to \mathsf{ff} && 37: && 0 \leqslant \mathsf{s}(y) \to \mathsf{tt} && 38: && \mathsf{s}(x) \leqslant \mathsf{s}(y) \to x \leqslant y\\
&39: && \mathsf{ff} \wedge \mathsf{ff} \to \mathsf{ff} && 40: && \mathsf{ff} \wedge \mathsf{tt} \to \mathsf{ff} && 41: && \mathsf{tt} \wedge \mathsf{ff} \to \mathsf{ff} && 42: && \mathsf{tt} \wedge \mathsf{tt} \to \mathsf{tt}\\
&43: && \mathsf{ff} \vee \mathsf{ff} \to \mathsf{ff} && 44: && \mathsf{ff} \vee \mathsf{tt} \to \mathsf{tt} && 45: && \mathsf{tt} \vee \mathsf{ff} \to \mathsf{tt} && 46: && \mathsf{tt} \vee \mathsf{tt} \to \mathsf{tt}\ .
\end{aligned}
$$

We set $\mathcal{R}_\mathsf{K} := \{(5)\text{—}(46)\}$. Let $\mathcal{F}_\mathsf{K}$ be the set of symbols occurring in $\mathcal{R}_\mathsf{K}$. Furthermore, let $\mathcal{T}_\mathsf{K} := \mathcal{T}_\mathsf{b}(\mathcal{D}_\mathsf{K}, \mathcal{C}_\mathsf{K})$ denote the set of basic terms where defined symbols $\mathcal{D}_\mathsf{K}$ coincide with the symbols defined by $\mathcal{R}_\mathsf{K}$, and $\mathcal{C}_\mathsf{K} := \mathcal{F}_\mathsf{K} \setminus \mathcal{D}_\mathsf{K}$. The problem $\langle \mathcal{R}_\mathsf{K}/\varnothing, \mathcal{R}_\mathsf{K}, \mathcal{T}_\mathsf{K}\rangle$ is the canonical innermost runtime complexity problem of $\mathcal{R}_\mathsf{K}$. The complexity of this problem is quadratic. The non-trivial proof relies on the invariant that the third argument P of kruskal denotes a partitioning of the initial set of nodes, at any time during reduction.

4.1. Suiting Reduction Orders to Complexity

Orders have been used quite early for the (automated) complexity analysis of rewrite systems. The seminal paper by Bonfante et al. [3] gives an early account on using reduction orders for complexity analysis, in the form of polynomial interpretations. Zankl and Korp [13] use pairs of orders $(\succsim, \succ)$, called *complexity pairs*, to estimate the derivational complexity in a relative setting. *Safe reduction pairs* [5] constitute a variation of complexity pairs. These are useful in conjunction with *dependency pairs*, compare Section 4.3. In the following, we introduce $\mathcal{P}$*-monotone complexity pairs*, which provide a unified account of these notions.

Fix a complexity problem $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$, and consider a reduction

$$t = t_0 \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} t_1 \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} t_2 \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} \cdots ,$$

for starting term $t \in \mathcal{T}$. Suppose we have shown termination of such sequences by means of a well founded order $\succ$ on terms: $t_i \succ t_{i+1}$ holds for all steps $t_i \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} t_{i+1}$. If there exists a function $f : \mathbb{N} \to \mathbb{N}$ which gives a bound on $\mathsf{dh}(t, \succ)$ expressed in the size of $t \in \mathcal{T}$, then f gives an upper bound on the complexity function of $\mathcal{P}$. However, not every order $\succ$ used in practice is finitely branching, and thus $\mathsf{dh}(t, \succ)$ is not necessarily well-defined. To account for such orders, Hirokawa and Moser [15] propose the notion of *G-collapsible order*. The following provides an adaptation of this notion.

Definition 4 (***G***-collapsible, Induced Complexity)**.** Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$ denote a complexity problem, consider an order $\succ$ on terms.

1. Suppose there exists a mapping $G : \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathbb{N}$ such that

   $$s \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} t \text{ and } s \succ t \implies G(s) > G(t) ,$$

   holds for all terms $s \in \to^*_\mathcal{P}(\mathcal{T})$. Then $\succ$ is called *G-collapsible* on $\mathcal{P}$. The order $\succ$ is *collapsible* with respect to $\mathcal{P}$ if there exists a mapping G such that $\succ$ is G-collapsible on $\mathcal{P}$.

2. Consider an order $\succ$ which is G-collapsible with respect to $\mathcal{P}$. Suppose that there exists a function $f : \mathbb{N} \to \mathbb{N}$ such that

   $$G(t) \in \mathcal{O}(f(|t|)) \text{ holds for all } t \in \mathcal{T}.$$

   Then we say that $\succ$ *induces* the complexity f on $\mathcal{P}$.

Lemma 1. *Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$ denote a complexity problem. Let $\succ$ be an order that is G-collapsible with respect to $\mathcal{P}$. Suppose*

$$s \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} t \implies s \succ t ,$$

holds for all $s \in \to^*_\mathcal{P}(\mathcal{T})$. Then for all terms $t \in \mathcal{T}$, $\mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}})$ is defined and $\mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}}) \leqslant G(t)$.

Proof. Immediate consequence of the assumptions and Definition 4. □

Consider an order $\succ$ that induces the complexity f on $\mathcal{P}$. If this order includes the relation $\xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}}$ on terms $t \in \to^*_\mathcal{P}(\mathcal{T})$, the above lemma shows that judgement $\vdash \mathcal{P}\colon f$ is valid. To check the inclusion, as in [13] we consider a pair of orders $(\succsim, \succ)$ on terms. Here $\succsim$ denotes a pre-order on terms, and $\succ$ an order compatible with $\succsim$: $\succsim \cdot \succ \cdot \succsim \subseteq \succ$. Zankl and Korp [13] further require that both orders are monotone and stable under substitutions. In this case, the assertions $\mathcal{W} \subseteq \succsim$ and $\mathcal{S} \subseteq \succ$ imply $\xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} \subseteq \succ$ as desired.

Guided by the observation that monotonicity is required only on argument positions t hat can be rewritten in reductions of starting terms, Hirokawa and Moser [16, 18] propose the use of μ*-monotone orders* for

runtime complexity analysis. In this context, the mapping μ is used to designate which arguments are *usable* in derivations, i.e. can be reduced. The following constitutes an adaptation of *usable replacement maps* from rewrite systems Hirokawa and Moser [18] to complexity problems. A *replacement map* on $\mathcal{F}$ is a mapping from function symbols $f/k \in \mathcal{F}$ to subsets of $\{1, \dots, k\}$. For a term t, the set $\mathcal{P}\mathrm{os}_\mu(t) \subseteq \mathcal{P}\mathrm{os}(t)$ of *μ-replacing positions* in t is defined such that $\mathcal{P}\mathrm{os}_\mu(t) := \{\epsilon\}$ if t is a variable, and $\mathcal{P}\mathrm{os}_\mu(t) := \{\epsilon\} \cup \{i{\cdot}p \mid i \in \mu(f) \text{ and } p \in \mathcal{P}\mathrm{os}_\mu(t_i)\}$ if $t = f(t_1, \dots, t_k)$. For a binary relation $\to$ on terms we denote by $\mathcal{T}_\mu(\to)$ the set of terms t where only sub-terms at μ-replacing positions are reducible: $t \in \mathcal{T}_\mu(\to)$ if for all positions $p \in \mathcal{P}\mathrm{os}(t)$, if $p \notin \mathcal{P}\mathrm{os}_\mu(t)$ then $t|_p \in \mathrm{NF}(\to)$.

Definition 5 (Usable Replacement Maps)**.** Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$ denote a complexity problem, and let $\mathcal{R}$ denote a set of rewrite rules. A replacement map μ is called a *usable replacement map* for $\mathcal{R}$ in $\mathcal{P}$, if $\to^*_{\mathcal{P}}(\mathcal{T}) \subseteq \mathcal{T}_\mu(\xrightarrow{\mathcal{Q}}_{\mathcal{R}})$.

Hence, a usable replacement map for a set of rules $\mathcal{R}$ approximates those positions at which a step due to $\mathcal{R}$ can happen, when considering $\mathcal{P}$-derivations starting from $\mathcal{T}$.

Example 5 (Continued from Example 3). Consider the $\mathcal{P}_\times$-derivation

$$\begin{aligned}\underline{\mathbf{2} \times \mathbf{1}} \to_{\mathcal{P}_\times} \mathbf{1} + (\underline{\mathbf{1} \times \mathbf{1}}) \to_{\mathcal{P}_\times} \mathbf{1} + (\underline{\mathbf{1} + (\mathbf{0} \times \mathbf{1})}) \to_{\mathcal{P}_\times} \mathbf{1} + \mathsf{s}(\mathbf{0} + (\underline{\mathbf{0} \times \mathbf{1}})) \\ \to_{\mathcal{P}_\times} \mathbf{1} + \mathsf{s}(\underline{\mathbf{0} + \mathbf{0}}) \to_{\mathcal{P}_\times} \underline{\mathbf{1} + \mathbf{1}} \to_{\mathcal{P}_\times} \mathsf{s}(\underline{\mathbf{0} + \mathbf{1}}) \to_{\mathcal{P}_\times} \mathbf{2}\,.\end{aligned}$$

Observe that if addition occurs in a context, then either under the second argument of addition or under the successor symbol. This holds even for all reductions of basic terms $\mathcal{T}_\times$. The map μ_+, defined by $\mu_+(\mathsf{s}) = \{1\}$, $\mu_+(+) = \{2\}$ and $\mu_+(\times) = \varnothing$, thus constitutes a usable replacement map for the addition rules $\{\mathtt{1}, \mathtt{2}\}$ in $\mathcal{P}_\times$.

An order $\succ$ on terms is called *μ-monotone* if it is monotone on μ-positions, in the sense that for all function symbols f, if $i \in \mu(f)$ and $s_i \succ t_i$ holds then $f(s_1, \dots, s_i, \dots, s_n) \succ f(s_1, \dots, t_i, \dots, s_n)$ holds.

Definition 6 ($\boldsymbol{\mathcal{P}}$-monotone, Complexity Pair)**.** Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$ denote a complexity problem.

1. A *complexity pair* $(\succsim, \succ)$ consists of a pre-order $\succsim$ and an order $\succ$ that are both closed under substitutions and satisfy $\succsim \cdot \succ \cdot \succsim \subseteq \succ$.
2. The complexity pair $(\succsim, \succ)$ is called $\mathcal{P}$-monotone if
   - $\succ$ is μ-monotone for a usable replacement map μ of $\mathcal{S}$ in $\mathcal{P}$; and
   - $\succsim$ is τ-monotone for a usable replacement map τ of $\mathcal{W}$ in $\mathcal{P}$.

Lemma 2. *Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$ be a complexity problem, and let $\mathcal{R} \subseteq \mathcal{S} \cup \mathcal{W}$ denote a set of rewrite rules in $\mathcal{P}$.*

1. *Let μ denote a usable replacement map for $\mathcal{R}$ in $\mathcal{P}$, and suppose $\mathcal{R} \subseteq \succ$ holds for a μ-monotone order $\succ$ that is stable under substitutions. Then*

$$s \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t \implies s \succ t\,,$$

   holds for all $s \in \to^*_{\mathcal{P}}(\mathcal{T})$.

2. *If $(\succsim, \succ)$ is a $\mathcal{P}$-monotone complexity pair such that $\mathcal{S} \subseteq \succ$ and $\mathcal{W} \subseteq \succsim$ holds, then*

$$s \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} t \implies s \succ t\,,$$

   holds for all $s \in \to^*_{\mathcal{P}}(\mathcal{T})$.

Proof. Consider the first assert. Since μ is a usable replacement map for $\mathcal{R}$ in $\mathcal{P}$, it suffices to show the claim for $s \in \mathcal{T}_\mu(\xrightarrow{\mathcal{Q}}_{\mathcal{R}})$. Suppose $s \xrightarrow{\mathcal{Q}}_{\mathcal{R},p} t$, hence $p \in \mathcal{P}\mathrm{os}_\mu(t)$. We show that for every prefix q of p, $s|_q \succ t|_q$ holds. The proof is by induction on $|p| - |q|$. The base case $q = p$ is covered by compatibility and stability under substitutions. For the inductive step, consider a prefix $q{\cdot}i$ of p, where by induction hypothesis $s|_{q\cdot i} \succ t|_{q\cdot i}$. Since $p \in \mathcal{P}\mathrm{os}_\mu(s)$ it is not difficult to see that $i \in \mu(f)$. Thus

$$s|_q = f(s_1, \ldots, s|_{q\cdot i}, \ldots, s_n) \succ f(s_1, \ldots, t|_{q\cdot i}, \ldots, s_n) = t|_q \;,$$

follows by μ-monotonicity of $\succ$ as desired. From this observation, the first assertion is obtained using $q = \epsilon$.

For the second assertion, consider a $\mathcal{Q}$-restricted relative step

$$s \xrightarrow{\mathcal{Q}}{}^*_{\mathcal{W}} \cdot \xrightarrow{\mathcal{Q}}_{\mathcal{S}} \cdot \xrightarrow{\mathcal{Q}}{}^*_{\mathcal{W}} t \;,$$

for $s \in \to^*_{\mathcal{P}}(\mathcal{T})$. Using the assumptions on $(\succsim, \succ)$ and the inclusions $\mathcal{W} \subseteq \succsim$ and $\mathcal{S} \subseteq \succ$ to satisfy the assumptions of the first assertion, we obtain $s \succsim^* \cdot \succ \cdot \succsim^* t$. Hence $s \succ t$ follows by transitivity of $\succsim$ and the inclusion $\succsim \cdot \succ \cdot \succsim \subseteq \succ$. □

As immediate consequence of Lemma 1 and Lemma 2, we obtain the following processor.

Theorem 2 (Complexity Pair Processor)**.** *Consider a $\mathcal{P}$-monotone complexity pair $(\succsim, \succ)$ such that $\succ$ induces the complexity f on $\mathcal{P}$. The following direct processor is sound:*

$$\frac{\mathcal{S} \subseteq \succ \quad \mathcal{W} \subseteq \succsim}{\vdash \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle \colon f}\ \mathsf{CP} \;.$$

Suppose no restriction is put on starting terms in $\mathcal{P}$. By definition, only the full replacement map is also a usable replacement map for rewrite rules occurring in $\mathcal{P}$. Then Theorem 2 requires that the orders $\succsim$ and $\succ$ are monotone in all argument positions. Hence on unconstrained sets of starting terms, our notion of $\mathcal{P}$-monotone complexity pair corresponds to the notion employed by Zankl and Korp [13].

4.2. Relative Decomposition

A variation of the complexity pair processor, that iteratively orients disjoint subsets of $\mathcal{S}$, occurred first in [13]. The following processor constitutes a straightforward generalisation of [13, Theorem 4.4] to our setting.

Theorem 3 (Decompose Processor)**.** *The following processor is sound:*

$$\frac{\vdash \langle \mathcal{S}_1/\mathcal{S}_2 \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle \colon f \quad \vdash \langle \mathcal{S}_2/\mathcal{S}_1 \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle \colon g}{\vdash \langle \mathcal{S}_1 \cup \mathcal{S}_2/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle \colon f + g}\ \mathsf{decompose} \;.$$

Here $f + g$ denotes the function h defined by $h(n) := f(n) + g(n)$.

Proof. The lemma follows from the inequality

$$\mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{S}_1 \cup \mathcal{S}_2/\mathcal{W}}) \leqslant \mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{S}_1/\mathcal{S}_2 \cup \mathcal{W}}) + \mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{S}_2/\mathcal{S}_1 \cup \mathcal{W}}) \;.$$ □

The decompose processor is a central ingredient for the automated complexity analysis. All participants of the recent complexity sub-division of the annual termination competition rely on variations of this processor for the combination of different proof techniques. In correspondence to the *rule removal processor* for termination analysis [32], one can combine Theorem 2 and Theorem 3. See [13] and [34] where a similar combination is proposed. This way the synthesis procedure implementing the complexity pair processor can determine a fitting partitioning of strict rules. Unlike in the rule removal processor for termination, for complexity analysis we need to keep the oriented rules in the weak component, cf. [13]. Oriented rules are thus *shifted* from the strict to the weak component. This is illustrated by the following example.

Example 6 (Continued from Example 5). Consider the linear polynomial interpretation $\mathcal{A}$ over $\mathbb{N}$ such that $0_{\mathcal{A}} = 0$, $\mathsf{s}_{\mathcal{A}}(x) = x$, $x +_{\mathcal{A}} y = y$ and $x \times_{\mathcal{A}} y = 1$. Let $\mathcal{P}_3 := \langle \{3\}/\{1,2,4\}, \mathcal{R}_\times, \mathcal{T}_\mathsf{b} \rangle$ denote the problem that accounts for the rule $3\colon\ 0 \times y \to 0$ in $\mathcal{P}_\times$. For a term t and assignment $\alpha : \mathcal{V} \to \mathbb{N}$, let $[\alpha]_{\mathcal{A}}(t)$ denote the interpretation of t by $\mathcal{A}$ defined in the obvious way. The induced order $>_{\mathcal{A}}$ together with the order $\geqslant_{\mathcal{A}}$, defined by $s \geqslant_{\mathcal{A}} t$ if $[\alpha]_{\mathcal{A}}(s) \geqslant [\alpha]_{\mathcal{A}}(t)$ holds for all assignments α, forms a $\mathcal{P}_3$-monotone complexity pair $(\geqslant_{\mathcal{A}}, >_{\mathcal{A}})$. Monotonicity can be shown using the replacement maps given in Example 5. The order $>_{\mathcal{A}}$ induces linear complexity on $\mathcal{P}_3$. According to Theorem 3, the following tree depicts a complexity proof $\langle \{1,2,4\}/\{3\}, \varnothing, \mathcal{T}_\mathsf{b} \rangle : g \vdash \mathcal{P}_\times : n + g$.

$$\dfrac{\dfrac{\{3\} \subseteq >_{\mathcal{A}} \quad \{1,2,4\} \subseteq \geqslant_{\mathcal{A}}}{\vdash \langle \{3\}/\{1,2,4\}, \varnothing, \mathcal{T}_\mathsf{b} \rangle : n}\ \mathsf{CP} \qquad \vdash \langle \{1,2,4\}/\{3\}, \varnothing, \mathcal{T}_\mathsf{b} \rangle : g}{\vdash \mathcal{P}_\times : n + g}\ \mathsf{decompose}.$$

The above complexity proof can now be completed iteratively, on the simpler problem $\langle \{1,2,4\}/\{3\}, \varnothing, \mathcal{T}_\mathsf{b} \rangle$. Since the complexity of $\mathcal{P}_\times$ is quadratic, one has to use a technique beyond linear polynomial interpretations here. The combination of complexity pairs with the decompose processor is formalised in the next Theorem.

Theorem 4 (Decompose CP Processor)**.** *Consider a $\mathcal{P}_1$-monotone complexity pair $(\succsim, \succ)$, for a complexity problem $\mathcal{P}_1 = \langle \mathcal{S}_1/\mathcal{S}_2 \cup \mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$. Suppose $\succ$ induces the complexity f on $\mathcal{P}_1$. The following processor is sound:*

$$\dfrac{\mathcal{S}_1 \subseteq \succ \quad \mathcal{W} \cup \mathcal{S}_2 \subseteq \succsim \qquad \vdash \langle \mathcal{S}_2/\mathcal{S}_1 \cup \mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle : g}{\vdash \langle \mathcal{S}_1 \cup \mathcal{S}_2/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle : f + g}\ \mathsf{decompose\ CP}\ .$$

Proof. Immediate consequence of Theorem 2 and Theorem 3. □

We remark that the decompose processor finds applications beyond its combination with complexity pairs, cf. Section 4.5.2.

4.3. Dependency Pairs for Complexity Analysis

For termination analysis, it is nowadays standard to transform a termination problem first into a *dependency pair* problem [36]. In essence, the dependency pairs of a TRS $\mathcal{R}$ reflect direct function calls in rules of $\mathcal{R}$. Termination of $\mathcal{R}$ is equivalent to the absence of infinite (and minimal) *dependency pair chains*, i.e. sequences of successive applications of dependency pairs, allowing modification of arguments using the original rules of $\mathcal{R}$. Reasoning over dependency pairs opens the door for a wealth of techniques. For an overview, we refer the reader to [32].

DPs track successive chains of function calls but forget contexts. The length of DP chains does in general not properly reflect the runtime complexity of the rewrite system under consideration [37]. To make the dependency pair technique applicable in the context of runtime complexity analysis, Hirokawa and Moser [5] introduced a variation of dependency pairs, the *weak dependency pairs*. Weak dependency pairs group various dependency pairs in a single rule, in order to simultaneously track multiple calls occurring in a rewrite rule. More recently, Noschinski et al. [34] introduced another variation of (weak) dependency pairs, *dependency tuples*, which overcome some deficiencies of the weak dependency pair approach. Dependency tuples are however only sound for analysing the innermost runtime complexity of TRSs. Modern termination tools like TCT thus implement both transformations.

In this section, we present the weak dependency pairs and dependency tuple transformation as complexity processors in our framework. We reprove the central theorems of Hirokawa and Moser [5] and Noschinski et al. [34], c.f. Theorem 5 and Theorem 6, respectively. This is necessary as neither Hirokawa and Moser [5] nor Noschinski et al. [34] treat relative rewriting. In contrast, our formulations allow us to cover complexity problems where the weak component in a complexity problem is not empty.

The following definition introduces the notion of *dependency pair complexity problem* (*DP problem* for short), which is liberal enough to capture the weak dependency pair and dependency tuple transformation.

Also, it allows us to express the various transformation techniques found in [5] and [34] that operate on weak dependendency pairs and dependency tuples, respectively. These transformations are covered in the following Section 4.4 and Section 4.5. As in [5, 34] we allow *compound symbols* in right hand sides of (grouped) dependency pairs. As for termination analysis, we consider derivations starting from *marked terms* only.

Definition 7 (Dependency Pairs, Dependency Pair Complexity Problem)**.** Let $\mathcal{F}$ be a signature with defined symbols $\mathcal{D}$.

1. For each $f/k \in \mathcal{D}$, let $f^\sharp$ denote a fresh function symbol of arity k, the *dependency pair symbol* (of f). The least extension of $\mathcal{F}$ to all dependency pair symbols is denoted by $\mathcal{F}^\sharp$.

   For a term $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ we define the *marking of* t as

   $$t^\sharp := \begin{cases} f^\sharp(t_1,\dots,t_k) & \text{if } t = f(t_1,\dots,t_k) \text{ and } f \in \mathcal{D}, \\ t & \text{otherwise.} \end{cases}$$

   For a set $T \subseteq \mathcal{T}(\mathcal{F},\mathcal{V})$, we denote by $T^\sharp$ the set of *marked terms* $T^\sharp = \{t^\sharp \mid t \in T\}$.

2. We denote by $\mathsf{Com} = \{\mathsf{c}^0_0, \mathsf{c}^1_0, \dots, \mathsf{c}^0_1, \mathsf{c}^1_1, \dots, \mathsf{c}^0_2, \mathsf{c}^1_2, \dots\}$ a countable infinite signature of constructor symbols, where the arity of c^i_k in Com is k for all $i,k \in \mathbb{N}$. Symbols in Com are called *compound symbols*.

3. Let $l, r, r_1, \dots, r_k \in \mathcal{T}(\mathcal{F},\mathcal{V})$ and $\mathsf{c}^i_k \in \mathsf{Com}$. A rewrite rule of the form $l^\sharp \to r^\sharp$ or $l^\sharp \to \mathsf{c}^i_k(r^\sharp_1,\dots,r^\sharp_k)$ is called a *grouped dependency pair*.

4. Let $\mathcal{S}$ and $\mathcal{W}$ be two TRSs over $\mathcal{T}(\mathcal{F},\mathcal{V})$, and let $\mathcal{S}^\sharp$ and $\mathcal{W}^\sharp$ be two sets of dependency pairs. A *dependency pair complexity problem*, or simply *DP problem*, is a runtime complexity problem $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp \cup \mathcal{S} / \mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$ over marked basic terms $\mathcal{T}^\sharp \subseteq \mathcal{T}^\sharp_{\mathsf{b}}(\mathcal{D},\mathcal{C})$.

We keep the convention that $\mathcal{R},\mathcal{S},\mathcal{W},\dots$ are TRSs over $\mathcal{T}(\mathcal{F},\mathcal{V})$, and the marked versions $\mathcal{R}^\sharp,\mathcal{S}^\sharp,\mathcal{W}^\sharp,\dots$ always denote sets of dependency pairs. We abbreviate the compound symbols c^0_k ($k \in \mathbb{N}$) by c_k. The identity of compound symbols occurring in terms is of no importance. This justifies that we write $\textsc{com}(t^\sharp_1,\dots,t^\sharp_k)$ for a term of the shape $\mathsf{c}^i_k(t^\sharp_1,\dots,t^\sharp_k)$ ($k,i \in \mathbb{N}$). For $k = 1$ we denote by $\textsc{com}(t^\sharp)$ also the term $t^\sharp$. Hence every dependency pair can be written as $l^\sharp \to \textsc{com}(r^\sharp_1,\dots,r^\sharp_k)$ for some terms $l, r_1, \dots, r_k \in \mathcal{T}(\mathcal{F},\mathcal{V})$.

Following [5, 34], our notion of (grouped) dependency pair is different to the notion of dependency pairs used in termination analysis [32, 36]. A grouped dependency pair $l^\sharp \to \textsc{com}(r^\sharp_1,\dots,r^\sharp_k)$ corresponds to the combination of multiple dependency pairs $l^\sharp \to r^\sharp_i$ ($i = 1,\dots,k$) obtained from the rule $l \to r$. Precisely which dependency pairs are combined depends on the used formalism, that is, whether we consider the setting of Hirokawa and Moser [5] or Noschinski et al. [34]. In the following, we call grouped dependency pairs simple *dependency pairs*, or *DPs* for short. No confusion can arise from this.

Example 7 (Continued from Example 3). Consider the DPs

$$\mathtt{47}\colon 0 +^\sharp y \to \mathsf{c}_0 \quad \mathtt{48}\colon \mathsf{s}(x) +^\sharp y \to x +^\sharp y \quad \mathtt{49}\colon 0 \times^\sharp y \to \mathsf{c}_0 \quad \mathtt{50}\colon \mathsf{s}(x) \times^\sharp y \to \mathsf{c}_2(x +^\sharp (x \times y), x \times^\sharp y)\ ,$$

denoted by $\mathcal{S}^\sharp_\times$ below. Let $\mathcal{T}^\sharp_\times$ be the set of marked basic terms with defined symbols $+^\sharp, \times^\sharp$ and constructors $\mathsf{s}, 0$. Then $\mathcal{P}^\sharp_{\times\text{-}\mathsf{i}} := \langle \mathcal{S}^\sharp_\times / \mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}^\sharp_\times\rangle$, where $\mathcal{R}_\times$ are the rules for addition and multiplication depicted in Example 3, is a DP problem. We anticipate that the DP problem $\mathcal{P}^\sharp_{\times\text{-}\mathsf{i}}$ reflects the complexity of the canonical *innermost* runtime complexity problem $\mathcal{P}^\sharp_{\times\text{-}\mathsf{i}}$ of $\mathcal{R}_\times$, compare Theorem 6 below.

Call an n-holed context C a *compound context* if it contains only compound symbols. Consider the $\mathcal{P}^\sharp_{\text{×-i}}$ derivation

$$\begin{aligned} D\colon\ \underline{\mathbf{2} \times^\sharp \mathbf{1}} &\to_{\mathcal{P}^\sharp_{\text{×-i}}} \mathsf{c}_2(\mathbf{1} +^\sharp (\underline{\mathbf{1} \times \mathbf{1}}), \mathbf{1} \times^\sharp \mathbf{1}) \\ &\to_{\mathcal{P}^\sharp_{\text{×-i}}} \mathsf{c}_2(\mathbf{1} +^\sharp (\mathbf{1} + (\underline{\mathbf{0} \times \mathbf{1}})), \mathbf{1} \times^\sharp \mathbf{1}) \\ &\to_{\mathcal{P}^\sharp_{\text{×-i}}} \cdots \\ &\to_{\mathcal{P}^\sharp_{\text{×-i}}} \mathsf{c}_2(\underline{\mathbf{1} +^\sharp \mathbf{1}}, \underline{\mathbf{1} \times^\sharp \mathbf{1}}) \\ &\to^2_{\mathcal{P}^\sharp_{\text{×-i}}} \mathsf{c}_2(\mathbf{0} +^\sharp \mathbf{1}, \mathsf{c}_2(\mathbf{1} +^\sharp (\mathbf{0} \times \mathbf{1}), \mathbf{0} \times^\sharp \mathbf{1}))\ . \end{aligned}$$

Observe that any term in the above sequence can be written as $C[t_1, \ldots, t_n]$ where C is a maximal compound context, and $t_1, \ldots, t_n$ are terms without compound symbols. For instance, the last term in this sequence is given as $C[\mathbf{0} \times^\sharp \mathbf{1}, \mathbf{1} +^\sharp (\mathbf{0} \times \mathbf{1}), \mathbf{0} \times^\sharp \mathbf{1}]$ for $C := \mathsf{c}_2(\Box, \mathsf{c}_2(\Box, \Box))$. This holds even in general. Note that the terms t_i $(i = 1, \ldots, n)$ are not necessarily marked, as our notion of dependency pair problem permits collapsing rule $l^\sharp \to x$ where x is a variable. We capture this observation with the set $\mathcal{T}^\sharp_\to$.

Definition 8. The set $\mathcal{T}^\sharp_\to$ is defined as the least set of terms such that

1. if $t \in \mathcal{T}(\mathcal{F})$ then $t \in \mathcal{T}^\sharp_\to$ and $t^\sharp \in \mathcal{T}^\sharp_\to$; and
2. if $t_1, \ldots, t_k \in \mathcal{T}^\sharp_\to$ and $\mathsf{c}_k \in \mathcal{C}\mathsf{om}$ then $\mathsf{c}_k(t_1, \ldots, t_k) \in \mathcal{T}^\sharp_\to$.

The simple observation can now be formalised as follows.

Lemma 3. *For every TRS $\mathcal{R}$ and DPs $\mathcal{R}^\sharp$, we have $\to^*_{\mathcal{R}^\sharp \cup \mathcal{R}}(\mathcal{T}^\sharp_\to) \subseteq \mathcal{T}^\sharp_\to$. In particular, $\to^*_{\mathcal{P}^\sharp}(\mathcal{T}^\sharp) \subseteq \mathcal{T}^\sharp_\to$ holds for every DP problem $\mathcal{P}^\sharp$ with starting terms $\mathcal{T}^\sharp$.*

Proof. The first half of the lemma follows by inductive reasoning. From this, the second half of the lemma follows, using that $\mathcal{T}^\sharp \subseteq \mathcal{T}^\sharp_\to$, taking $\mathcal{R}^\sharp := \mathcal{S}^\sharp \cup \mathcal{W}^\sharp$ and $\mathcal{R} := \mathcal{S} \cup \mathcal{W}$. □

4.3.1. Weak Dependency Pairs

To analyse the runtime complexity of TRS $\mathcal{R}$, Hirokawa and Moser [5] group parallel function calls (and variables) of a rule $l \to r$ in a (weak) DP $l^\sharp \to \mathsf{c}_k(r^\sharp_1, \ldots, r^\sharp_k)$. This is made precise in the following definition.

Definition 9 (Weak Dependency Pairs [5])**.** Let $\mathcal{R}$ denote a TRS.

1. Consider a rule $l \to C[r_1, \ldots, r_k]$ in $\mathcal{R}$, where C is a maximal context containing only constructors. We define
$$\mathsf{WDP}(l \to r) := l^\sharp \to \textsc{com}(r^\sharp_1, \ldots, r^\sharp_k)\ ,$$
and call $\mathsf{WDP}(l \to r)$ the *weak dependency pair* of $l \to r$.
2. The *weak dependency pairs* $\mathsf{WDP}(\mathcal{R})$ of a TRS $\mathcal{R}$ are given by
$$\mathsf{WDP}(\mathcal{R}) := \{\mathsf{WDP}(l \to r) \mid l \to r \in \mathcal{R}\}\ .$$

Notice that $\mathsf{WDP}(\mathcal{R})$ is unique up to renaming of compound symbols.[4] Also note that for $l \to r \in \mathcal{R}$, the left-hand side $l^\sharp$ is always a marked term $f^\sharp(l_1, \ldots, l_n)$ with $f^\sharp$ a dependency pair symbol.

Example 8 (Continued from Example 3). Consider the TRS $\mathcal{R}_\times$ given in Example 3. Then $\mathsf{WDP}(\mathcal{R}_\times)$ consists of the following four rules:

$$\texttt{51}\colon\ 0 +^\sharp y \to y \qquad \texttt{52}\colon\ \mathsf{s}(x) +^\sharp y \to x +^\sharp y \qquad \texttt{53}\colon\ 0 \times^\sharp y \to y \qquad \texttt{54}\colon\ \mathsf{s}(x) \times^\sharp y \to x +^\sharp (x \times y)\ .$$

[4] In our implementation, we have chosen to assign for each rule a fresh compound symbol.

In [5] it is shown that for any term $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$, $\mathsf{dh}(t,\to_{\mathcal{R}}) =_{\mathsf{k}} \mathsf{dh}(t^\sharp,\to_{\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}})$ holds. We extend this result to our setting. The following lemma serves as a preparatory step.

Lemma 4. *Let $\mathcal{R}$ and $\mathcal{Q}$ be two TRSs. Then every derivation*

$$t = t_0 \xrightarrow{\mathcal{Q}}_{\mathcal{R}_1} t_1 \xrightarrow{\mathcal{Q}}_{\mathcal{R}_2} t_2 \xrightarrow{\mathcal{Q}}_{\mathcal{R}_3} \cdots ,$$

for a basic term t and $\mathcal{R}_i \subseteq \mathcal{R}$ $(i \geqslant 1)$ is simulated step-wise by a derivation

$$t^\sharp = s_0 \xrightarrow{\mathcal{Q}}_{\mathsf{WDP}(\mathcal{R}_1)\cup\mathcal{R}_1} s_1 \xrightarrow{\mathcal{Q}}_{\mathsf{WDP}(\mathcal{R}_2)\cup\mathcal{R}_2} s_2 \xrightarrow{\mathcal{Q}}_{\mathsf{WDP}(\mathcal{R}_3)\cup\mathcal{R}_3} \cdots ,$$

and vice versa.

Proof. For a term s, let $P(s) \subseteq \mathcal{P}\mathrm{os}_{\mathcal{D}}(s)$ denote the set of *minimal* positions p with $\mathsf{rt}(s|_p)$ a defined symbol. Call a term $u = C[u_1,\ldots,u_n]$ *good for* s if C is a maximal context containing only constructors (including compound symbols) and there exists an injective mapping $m : P(s) \to \mathcal{P}\mathrm{os}_{\Box}(C)$ such that $u|_{m(p)} = s|_p$ or $u|_{m(p)} = (s|_p)^\sharp$. Observe that to every $\mathcal{R}$-redex $s|_q$ in s the mapping m associates a $\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}$-reducible subterm $u|_{m(p)}$ of u. Here p is the (unique) prefix of q with $p \in P(s)$.

Consider a step $s \xrightarrow{\mathcal{Q}}_{\{l\to r\},q} t$ for $l \to r \in \mathcal{R}$, and suppose $\mathsf{WDP}(l \to r) = l^\sharp \to \textsc{com}(r_1^\sharp,\ldots,r_m^\sharp)$. Fix a term $u = C[u_1,\ldots,u_n]$ with C a maximal constructor context that is good for s, as witnessed by a mapping $m_s : P(s) \to \mathcal{P}\mathrm{os}_{\Box}(C)$. We show that there exists a term v with

$$u \xrightarrow{\mathcal{Q}}_{\{\mathsf{WDP}(l\to r),l\to r\}} v \qquad \text{and } v \text{ is good for } t.$$

This property establishes the simulation from left to right. Let $p \in P(s)$ denote the unique prefix of the redex position q. We consider the cases $q < p$ and $q = p$ separately.

Consider first the case $q < p$. Then

$$u = C[u_1,\ldots,u_i,\ldots,u_n] \xrightarrow{\mathcal{Q}}_{\{l\to r\}} C[u_1,\ldots,v_i,\ldots,u_n] ,$$

for $u_i = u|_{m(p)}$ and v_i possible marked versions of $s|_p$ and $t|_p$, respectively. Let $v := C[u_1,\ldots,v_i,\ldots,v_n]$. By assumption $p \in P(s)$ the root of $t|_p$, viz the root of $s|_p$, is defined. Thus $P(s) = P(t)$ and the mapping m_s witnesses also that v is good for t.

Now consider the case where $q = p$. Then $s|_p = l\sigma$ and $t|_p = r\sigma$ for some substitution σ. We distinguish again two sub-cases. First, suppose that $u_i = u|_{m(p)} = (s|_p)^\sharp$ is marked. Thus

$$u = C[u_1,\ldots,l^\sharp\sigma,\ldots,u_n] \xrightarrow{\mathcal{Q}}_{\mathsf{WDP}(l\to r)} C[u_1,\ldots,\textsc{com}(r_1^\sharp\sigma,\ldots,r_m^\sharp\sigma),\ldots,u_n] ,$$

by definition of $\mathsf{WDP}(l \to r)$. Let $v := C[u_1,\ldots,\textsc{com}(r_1^\sharp\sigma,\ldots,r_m^\sharp\sigma),\ldots,u_n]$, and denote by C_r the maximal constructor context of $\textsc{com}(r_1^\sharp\sigma,\ldots,r_m^\sharp\sigma)$. Then it is not difficult to construct an injective mapping $m_r : P(r\sigma) \to \mathcal{P}\mathrm{os}_{\Box}(C_r)$ which verifies that $\textsc{com}(r_1^\sharp\sigma,\ldots,r_m^\sharp\sigma)$ is good for $r\sigma$. Observe $P(t) = (P(s)\setminus\{p\})\cup\{p\cdot q_i \mid q_i \in P(r\sigma)\}$ and that all positions in $P(t)$ are parallel. Hence the injective mapping m_t, given by

$$m_t(q) := \begin{cases} m_s(p)\cdot m_r(q_i) & \text{if } q = p\cdot q_i, \\ m_s(q) & \text{if } q \parallel p. \end{cases}$$

is well-defined for every $q \in \mathcal{P}\mathrm{os}(t)$, and verifies that v is good for t.

Finally, suppose that $u|_{m(p)}$ is not marked, that is $u_i = l\sigma$ for some $i \in \{1,\ldots,n\}$. Then

$$u = C[u_1,\ldots,l\sigma,\ldots,u_n] \xrightarrow{\mathcal{Q}}_{\{l\to r\}} C[u_1,\ldots,r\sigma,\ldots,u_n] .$$

Let $v := C[u_1,\ldots,r\sigma,\ldots,u_n]$. Then v is good for t, following the pattern of the previous case, exploiting that $r\sigma$ is trivially good for itself.

For the direction from right to left, consider a term $u = C[u_1,\ldots,u_n]$ where C is a compound context, and u_i $(i = 1,\ldots,n)$ denote possibly marked terms without compound symbols. Call a term $s \in \mathcal{T}(\mathcal{F},\mathcal{V})$

good for u if s is obtained from u by unmarking symbols, and replacing C with a context consisting only of constructors. By case analysis on $u \xrightarrow{\mathcal{Q}}_{\mathsf{WDP}(\mathcal{R}^\sharp)\cup\mathcal{R}} v$, it can be verified that for any such u, if s is good for u then there exists a term t with $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t$ that is good for v. Since the starting term $t^\sharp$ is trivially of the considered shape, the simulation follows. □

Theorem 5 (Weak Dependency Pair Processor)**.** *Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$ denote a runtime complexity problem. The following processor is sound and complete.*

$$\frac{\vdash \langle \mathsf{WDP}(\mathcal{S}) \cup \mathcal{S}/\mathsf{WDP}(\mathcal{W}) \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle \colon f}{\vdash \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle \colon f}\ \mathsf{WDP}\ .$$

Proof. Let $\mathcal{P}^\sharp := \langle \mathsf{WDP}(\mathcal{S}) \cup \mathcal{S}/\mathsf{WDP}(\mathcal{W}) \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$. Suppose first $\mathsf{cp}_{\mathcal{P}^\sharp} \in \mathcal{O}(f(n))$. Lemma 4 shows that every $\to_{\mathcal{P}}$ reduction of $t \in \mathcal{T}$ is simulated by a corresponding $\to_{\mathcal{P}^\sharp}$ reduction starting from $t^\sharp \in \mathcal{T}^\sharp$. Observe that every $\xrightarrow{\mathcal{Q}}_{\mathcal{S}}$ step in the considered derivation is simulated by a $\xrightarrow{\mathcal{Q}}_{\mathsf{WDP}(\mathcal{S})\cup\mathcal{S}}$ step. We thus obtain $\mathsf{cp}_{\mathcal{P}} \in \mathcal{O}(f(n))$. This proves soundness, completeness is obtained dual. □

Observe that the generated sub-problem is a DP complexity problem as given by Definition 7. Notice also that when the input is an innermost complexity problem, then so is the obtained DP problem.

Example 9 (Continued from Example 3 and 8). Reconsider the TRS $\mathcal{R}_\times$ given in Example 3, together with $\mathsf{WDP}(\mathcal{R}_\times)$ depicted in Example 8. According to the weak dependency pair processor, the inference

$$\frac{\vdash \langle \mathsf{WDP}(\mathcal{R}_\times) \cup \mathcal{R}_\times/\varnothing, \varnothing, \mathcal{T}_\times^\sharp\rangle \colon n^2}{\vdash \langle \mathcal{R}_\times/\varnothing, \varnothing, \mathcal{T}_\times\rangle \colon n^2}\ \mathsf{WDP}\ .$$

is sound and complete.

The change in signature often makes the generated sub-problem easier to analyse. In particular, the generated sub-problem is amenable to many of the processors suited for dependency pair problems introduced below.

4.3.2. Dependency Tuples

Consider a DP problem of the form $\langle \mathcal{S}^\sharp/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$. The analysis of this problem requires only an estimation of applications of DPs, which are applied in compound contexts only. This property makes the analysis considerably simpler. Some processors tailored for DP problems are even sound only in this setting, for instance *(safe) reduction pairs* (cf. Definition 11 below) and various syntactic simplifications proposed in Section 4.5. In contrast to weak dependency pairs, *dependency tuples* [34] allow the translation of an *innermost* runtime complexity problem directly into a DP problem of this simpler form. This approach, however, cannot guarantee completeness.

Definition 10 (Dependency Tuples [34])**.** Let $\mathcal{R}$ denote a TRS.

1. Consider a rule $l \to r$ in $\mathcal{R}$, and let $r_1, \dots, r_k$ denote *all* sub-terms of the right-hand side whose root symbol is a defined symbol. We define

$$\mathsf{DT}(l \to r) := l^\sharp \to \textsc{com}(r_1^\sharp, \dots, r_k^\sharp)\ ,$$

   and call $\mathsf{DT}(l \to r)$ the *dependency tuple* of $l \to r$.

2. The *dependency tuples* $\mathsf{DT}(\mathcal{R})$ of a TRS $\mathcal{R}$ are given by

$$\mathsf{DT}(\mathcal{R}) := \{\mathsf{DT}(l \to r) \mid l \to r \in \mathcal{R}\}\ .$$

Example 10 (Continued from Example 7). The four DPs (47)–(50) depicted in Example 7 constitute the dependency tuples of $\mathcal{R}_\times$ from Example 3.

The central theorem of [34] states that dependency tuples are sound for innermost runtime complexity analysis. We extend this result to a relative setting.

Lemma 5. *Let $\mathcal{R}$ and $\mathcal{Q}$ be two TRSs such that* $\mathrm{NF}(\mathcal{Q}) \subseteq \mathrm{NF}(\mathcal{R})$. *Then every derivation*

$$t = t_0 \xrightarrow{\mathcal{Q}}_{\mathcal{R}_1} t_1 \xrightarrow{\mathcal{Q}}_{\mathcal{R}_2} t_2 \xrightarrow{\mathcal{Q}}_{\mathcal{R}_3} \cdots ,$$

for a basic term t and $\mathcal{R}_i \subseteq \mathcal{R}$ $(i \geqslant 1)$ is simulated step-wise by a derivation

$$t^\sharp = s_0 \xrightarrow{\mathcal{Q}}_{\mathsf{DT}(\mathcal{R}_1)/\mathcal{R}_1} s_1 \xrightarrow{\mathcal{Q}}_{\mathsf{DT}(\mathcal{R}_2)/\mathcal{R}_2} s_2 \xrightarrow{\mathcal{Q}}_{\mathsf{DT}(\mathcal{R}_3)/\mathcal{R}_3} \cdots .$$

Proof. The proof follows the pattern of the proof of Lemma 4. Define $P(s)$ as the restriction of $\mathcal{P}\mathrm{os}(s)$ to reducible subterms in s whose root symbol is defined: $P(s) := \{p \mid \mathsf{rt}(s|_p) \in \mathcal{D} \text{ and } s|_p \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t \text{ for some term } t\}$. Call a term u *good for* s if $u = C[u_1^\sharp, \ldots, u_n^\sharp]$ for a context C and $u_1^\sharp, \ldots, u_n^\sharp$ are marked version of the subterms $s|_p$ for all $p \in P(s)$. More precisely, there exists an injective function $m : P(s) \to \mathcal{P}\mathrm{os}_\Box(C)$ such that $u|_{m(p)} = (s|_p)^\sharp$ for every position $p \in P(s)$ holds.

Consider a rewrite step $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}_i} t$ at position p, let $l \to r \in \mathcal{R}_i$ denote the rewrite rule applied in this step, and let σ be the substitution with $l\sigma = s|_p$. Consider a term $u = C[u_1^\sharp, \ldots, u_n^\sharp]$ good for s, as witnessed by a mapping $m_s : P(s) \to \mathcal{P}\mathrm{os}_\Box(C)$. Observe that $p \in P(s)$ and thus for $\mathsf{DT}(l \to r) = l^\sharp \to \mathrm{COM}(r_1^\sharp, \ldots, r_m^\sharp)$,

$$u = C[u_1^\sharp, \ldots, l^\sharp\sigma, \ldots, u_n^\sharp] \xrightarrow{\mathcal{Q}}_{\mathsf{DT}(l\to r)} C[u_1^\sharp, \ldots, \mathrm{COM}(r_1^\sharp\sigma, \ldots, r_m^\sharp\sigma), \ldots, u_n^\sharp] .$$

Let $C' := C[\Box, \ldots, \mathrm{COM}(\Box, \ldots, \Box), \ldots, \Box]$ and let $v' := C'[u_1^\sharp, \ldots, r_1^\sharp\sigma, \ldots, r_m^\sharp\sigma, \ldots, u_n^\sharp]$.

Observe that $P(t) \subseteq \{q \mid q \in P(s) \text{ with } q \parallel p \text{ or } q < p\} \cup \{p{\cdot}q \mid q \in \mathcal{P}\mathrm{os}_\mathcal{D}(r)\}$. This follows as the substitution σ maps variables to $\mathrm{NF}(\mathcal{Q}) \subseteq \mathrm{NF}(\mathcal{R})$. Consider a position $q \in P(t)$ with $q < p$, hence $q \in P(s)$, and $j \in \{1, \ldots, n\}$ be such that $u_j^\sharp = (s|_q)^\sharp$. Note that j exists as u is good for s. Since the rewrite position p in $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}_i} t$ is strictly below q, we have $u_j^\sharp \xrightarrow{\mathcal{Q}}_{\mathcal{R}_i} (t|_q)^\sharp$. Define v as the term obtained from v' by rewriting all terms $u_j^\sharp = v|_{m_s(q)}$ for $q < p$ in this way. Thus $\xrightarrow{\mathcal{Q}}_{\mathsf{DT}(\mathcal{R}_i)/\mathcal{R}_i} v$, it remains to show that v is good for t.

Consider a position $q \in P(t)$ with $q \geqslant p$. By the observation on $P(t)$, $q = p{\cdot}q_j$ with $q_j \in \mathcal{P}\mathrm{os}_\mathcal{D}(r)$, and thus $r|_{q_j} = r_j$ for some $j \in \{1, \ldots, m\}$. We denote by q' the position of $r_j^\sharp$ in the right-hand side $\mathrm{COM}(r_1^\sharp, \ldots, r_m^\sharp)$ of $\mathsf{DT}(l \to r)$. Then the following injective mapping $m_t : P(t) \to \mathcal{P}\mathrm{os}_\Box(C')$, defined by

$$m_t(q) := \begin{cases} m_s(p){\cdot}q' & \text{if } q \geqslant p, \\ m_s(q) & \text{if } q \parallel p \text{ or } q < p, \end{cases}$$

witnesses that v is good for t: if $q < p$ then we already observed $v|_{m_t(q)} = v|_{m_s(q)} = (t|_q)^\sharp$. If $q \parallel p$ then $t|_q = s|_q$ and $v|_{m_t(q)} = u|_{m_s(q)}$ and thus $v|_{m_t(q)} = (s|_t)^\sharp$ as desired. Finally, consider $q \geqslant p$, i.e. $q = p{\cdot}q_j$ for some $q_j \in \mathcal{P}\mathrm{os}_\mathcal{D}(r)$ and set $r_j := r|_{q_j}$. Then $(t|_q)^\sharp = (r_j\sigma)^\sharp = r_j^\sharp\sigma = v|_{m_s(p)\cdot q'} = v|_{m_t(p)}$, which concludes the final case. □

Theorem 6 (Dependency Tuple Processor)**.** *Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$ denote an* innermost *runtime complexity problem. The following processor is sound.*

$$\frac{\vdash \langle \mathsf{DT}(\mathcal{S})/\mathsf{DT}(\mathcal{W}) \cup \mathcal{S} \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle \colon f}{\vdash \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle \colon f}\ \mathsf{DT} .$$

Proof. The theorem follows in correspondence to Theorem 5, replacing the application of Lemma 4 with Lemma 5. □

We emphasise that for $\mathcal{W} = \varnothing$, Theorem 6 corresponds to [34, Theorem 10]. The following example clarifies that the restriction to innermost rewriting is necessary.

Example 11. Consider the constructor TRS $\mathcal{R}_{\mathsf{dup}}$ given by the following rules:

$$55\colon\ \mathsf{btree}(0) \to \mathsf{leaf} \qquad 56\colon\ \mathsf{btree}(\mathsf{s}(n)) \to \mathsf{dup}(\mathsf{btree}(n)) \qquad 57\colon\ \mathsf{dup}(t) \to \mathsf{c}(t,t)\ ,$$

that computes a binary tree of height n on input $\mathsf{s}^n(0)$. This TRS admits exponential long outermost reductions, obtained by successively duplicating redexes. The dependency tuples $\mathsf{DT}(\mathcal{R}_{\mathsf{dup}})$ are given by the three rules

$$58\colon\ \mathsf{btree}^\sharp(0) \to \mathsf{c}_0 \qquad 59\colon\ \mathsf{btree}^\sharp(\mathsf{s}(n)) \to \mathsf{c}_2(\mathsf{dup}^\sharp(\mathsf{btree}(n)), \mathsf{btree}^\sharp(n)) \qquad 60\colon\ \mathsf{dup}^\sharp(t) \to \mathsf{c}_0\ .$$

It is not difficult to see that in a $\mathsf{DT}(\mathcal{R}_{\mathsf{dup}}) \cup \mathcal{R}_{\mathsf{dup}}$ derivation starting from $\mathsf{btree}^\sharp(\mathbf{n})$ ($n \in \mathbb{N}$), the overall number of applications of a dependency pair is bounded linearly in n, i.e. $\vdash \langle \mathsf{DT}(\mathcal{R}_{\mathsf{dup}})/\mathcal{R}_{\mathsf{dup}}, \varnothing, \mathcal{T}_{\mathsf{b}}^\sharp\rangle \colon n$. Permitting the inference

$$\frac{\vdash \langle \mathsf{DT}(\mathcal{R}_{\mathsf{dup}})/\mathcal{R}_{\mathsf{dup}}, \varnothing, \mathcal{T}_{\mathsf{b}}^\sharp\rangle \colon n}{\vdash \langle \mathcal{R}_{\mathsf{dup}}/\varnothing, \varnothing, \mathcal{T}_{\mathsf{b}}\rangle \colon n}$$

would allow us to wrongly deduce that the runtime complexity of $\mathcal{R}_{\mathsf{dup}}$ is linear.

Example 12 (Continued from Example 4). The following inference starts the proof of quadratic innermost runtime complexity of the TRS $\mathcal{R}_{\mathsf{K}}$ given in Example 4. We transform its canonical innermost complexity problem into a DP problem using dependency tuples (Theorem 6).

$$\frac{\vdash \langle \mathcal{S}_{\mathsf{K}}^\sharp/\mathcal{R}_{\mathsf{K}}, \mathcal{R}_{\mathsf{K}}, \mathcal{T}_{\mathsf{K}}^\sharp\rangle \colon n^2}{\vdash \langle \mathcal{R}_{\mathsf{K}}/\varnothing, \mathcal{R}_{\mathsf{K}}, \mathcal{T}_{\mathsf{K}}\rangle \colon n^2}\ \mathsf{DT}\ .$$

Here $\mathcal{S}_{\mathsf{K}}^\sharp := \mathsf{DT}(\mathcal{R}_{\mathsf{K}})$ consists of the DPs depicted in Figure 2.

4.3.3. Reduction Pairs

Reduction pairs were introduced in the context of termination analysis [36]. *Safe reduction pairs* [5], aka COM-*monotone* reduction pairs [34], constitute a variation that accounts for compound symbols in complexity problems.

Definition 11 (Safe Reduction Pair [5])**.** A *reduction pair* $(\succsim, \succ)$ consists of a rewrite preorder $\succsim$ and a compatible well-founded order $\succ$ which is closed under substitutions. Here compatibility means that the inclusion $\succsim \cdot \succ \cdot \succsim \subseteq \succ$ holds. The reduction pair $(\succsim, \succ)$ is called *safe* if the order $\succ$ is monotone in all coordinates on compound symbols, i.e. for every $\mathsf{c}_k \in \mathcal{C}\mathsf{om}$, for all $i = 1, \dots, k$ and terms $s_1, \dots, s_k, s_i'$, if $s_i \succ s_i'$ then $\mathsf{c}_k(s_1, \dots, s_i, \dots, s_k) \succ \mathsf{c}_k(s_1, \dots, s_i', \dots, s_k)$ holds.

Proposition 1 ([5])**.** *Let $(\succsim, \succ)$ be a safe reduction pair, let $\mathcal{S}^\sharp$ be a set of weak dependency pairs and let $\mathcal{W}$ be a rewrite system. If $\mathcal{S}^\sharp \subseteq \succ$ and $\mathcal{W} \subseteq \succsim$ then $\mathsf{dh}(t, \to_{\mathcal{S}^\sharp/\mathcal{W}}) \leqslant f(t)$ where $f : \mathbb{N} \to \mathbb{N}$ is the complexity induced by $\succ$.*

The above proposition refers to the application of safe reduction pairs in the main theorem of Hirokawa and Moser [5]. Together with the following simple observation, this proposition is a straightforward consequence of our complexity pair processor (Theorem 2)

Lemma 6. *Let $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$ be a DP problem such that the strict component contains no rewrite rules.*

1. *Suppose μ denotes a usable replacement map for dependency pairs $\mathcal{R}^\sharp$ in $\mathcal{P}^\sharp$. Then μ_{COM} is a usable replacement map for $\mathcal{R}^\sharp$ in $\mathcal{P}^\sharp$. Here μ_{COM} denotes the restriction of μ to compound symbols in the following sense: $\mu_{\text{COM}}(c) := \mu(c)$ for all $c \in \mathcal{C}\mathsf{om}$, and otherwise $\mu_{\text{COM}}(f) := \varnothing$ for $f \in \mathcal{F}^\sharp$.*

$$
\begin{array}{lll}
61\colon \mathsf{src}^\sharp((n,w,m)) \to \mathsf{c}_0 & 62\colon \mathsf{wt}^\sharp((n,w,m)) \to \mathsf{c}_0 & 63\colon \mathsf{trg}^\sharp((n,w,m)) \to \mathsf{c}_0
\end{array}
$$

$$
\begin{array}{rl}
64\colon & \mathsf{forest}^\sharp(\mathsf{graph}(N,E)) \to \mathsf{c}_3(\mathsf{kruskal}^\sharp(\mathsf{sort}(E),[\,],\mathsf{partitions}(N)),\mathsf{sort}^\sharp(E),\mathsf{partitions}^\sharp(N)) \\
65\colon & \mathsf{partitions}^\sharp([\,]) \to \mathsf{c}_0 \\
66\colon & \mathsf{partitions}^\sharp(n :: N) \to \mathsf{partitions}^\sharp(N) \\
67\colon & \mathsf{kruskal}^\sharp([\,],W,P) \to \mathsf{c}_0 \\
68\colon & \mathsf{kruskal}^\sharp(e :: E,W,P) \to \mathsf{c}_2(\mathsf{kruskal?}^\sharp(\mathsf{inBlock}(e,P),e,E,W,P),\mathsf{inBlock}^\sharp(e,P)) \\
69\colon & \mathsf{kruskal?}^\sharp(\mathsf{tt},e,E,W,P) \to \mathsf{kruskal}^\sharp(E,W,P) \\
70\colon & \mathsf{kruskal?}^\sharp(\mathsf{ff},e,E,W,P) \to \mathsf{c}_2(\mathsf{kruskal}^\sharp(E,e :: W,\mathsf{join}(e,P,[\,])),\mathsf{join}^\sharp(e,P,[\,])) \\
71\colon & \mathsf{inBlock}^\sharp(e,[\,]) \to \mathsf{c}_0 \\
72\colon & \mathsf{inBlock}^\sharp(e,p :: P) \to \mathsf{c}_7((\mathsf{src}(e) \in p \wedge \mathsf{trg}(e) \in p) \vee^\sharp \mathsf{inBlock}(e,P),\mathsf{src}(e) \in p \wedge^\sharp \mathsf{trg}(e) \in p, \\
& \qquad \mathsf{src}(e) \in^\sharp p,\mathsf{trg}(e) \in^\sharp p,\mathsf{src}^\sharp(e),\mathsf{trg}(e)^\sharp,\mathsf{inBlock}^\sharp(e,P)) \\
73\colon & \mathsf{join}^\sharp(e,[\,],q) \to \mathsf{c}_0 \\
74\colon & \mathsf{join}^\sharp(e,p :: P,q) \to \mathsf{c}_6(\mathsf{join?}^\sharp(\mathsf{src}(e) \in p \vee \mathsf{trg}(e) \in p,e,p,P,q),\mathsf{src}(e) \in p \vee^\sharp \mathsf{trg}(e) \in p, \\
& \qquad \mathsf{src}(e) \in^\sharp p,\mathsf{trg}(e) \in^\sharp p,\mathsf{src}^\sharp(e),\mathsf{trg}(e)^\sharp) \\
75\colon & \mathsf{join?}^\sharp(\mathsf{tt},e,p,P,q) \to \mathsf{c}_2(\mathsf{join}^\sharp(e,P,p +\!\!+ q),p +\!\!+^\sharp q) \\
76\colon & \mathsf{join?}^\sharp(\mathsf{ff},e,p,P,q) \to \mathsf{join}^\sharp(e,P,q) \\
77\colon & \mathsf{sort}^\sharp([\,]) \to \mathsf{c}_0 \\
78\colon & \mathsf{sort}^\sharp(e :: E) \to \mathsf{c}_2(\mathsf{insert}^\sharp(e,\mathsf{sort}(E)),\mathsf{sort}^\sharp(E)) \\
79\colon & \mathsf{insert}^\sharp(e,[\,]) \to \mathsf{c}_0 \\
80\colon & \mathsf{insert}^\sharp(e,f :: E) \to \mathsf{c}_4(\mathsf{insert?}^\sharp(\mathsf{wt}(e) \leqslant \mathsf{wt}(f),e,f,E),\mathsf{wt}(e) \leqslant^\sharp \mathsf{wt}(f),\mathsf{wt}^\sharp(e),\mathsf{wt}^\sharp(f)) \\
81\colon & \mathsf{insert?}^\sharp(\mathsf{tt},e,f,E) \to \mathsf{c}_0 \\
82\colon & \mathsf{insert?}^\sharp(\mathsf{ff},e,f,E) \to \mathsf{insert}^\sharp(e,E) \\
83\colon & n \in^\sharp [\,] \to \mathsf{c}_0 \\
84\colon & n \in^\sharp (m :: p) \to \mathsf{c}_3(n = m \vee^\sharp n \in p,n =^\sharp m,n \in^\sharp p) \\
85\colon & [\,] +\!\!+^\sharp q \to \mathsf{c}_0 \\
86\colon & (n :: p) +\!\!+^\sharp q \to p +\!\!+^\sharp q
\end{array}
$$

$$
\begin{array}{llll}
87\colon\ 0 =^\sharp 0 \to \mathsf{c}_0 & 88\colon\ \mathsf{s}(x) =^\sharp 0 \to \mathsf{c}_0 & 89\colon\ 0 =^\sharp \mathsf{s}(y) \to \mathsf{c}_0 & 90\colon\ \mathsf{s}(x) =^\sharp \mathsf{s}(y) \to x =^\sharp y \\
91\colon\ 0 \leqslant^\sharp 0 \to \mathsf{c}_0 & 92\colon\ \mathsf{s}(x) \leqslant^\sharp 0 \to \mathsf{c}_0 & 93\colon\ 0 \leqslant^\sharp \mathsf{s}(y) \to \mathsf{c}_0 & 94\colon\ \mathsf{s}(x) \leqslant^\sharp \mathsf{s}(y) \to x \leqslant^\sharp y \\
95\colon\ \mathsf{ff} \wedge^\sharp \mathsf{ff} \to \mathsf{c}_0 & 96\colon\ \mathsf{ff} \wedge^\sharp \mathsf{tt} \to \mathsf{c}_0 & 97\colon\ \mathsf{tt} \wedge^\sharp \mathsf{ff} \to \mathsf{c}_0 & 98\colon\ \mathsf{tt} \wedge^\sharp \mathsf{tt} \to \mathsf{c}_0 \\
99\colon\ \mathsf{ff} \vee^\sharp \mathsf{ff} \to \mathsf{c}_0 & 100\colon\ \mathsf{ff} \vee^\sharp \mathsf{tt} \to \mathsf{c}_0 & 101\colon\ \mathsf{tt} \vee^\sharp \mathsf{ff} \to \mathsf{c}_0 & 102\colon\ \mathsf{tt} \vee^\sharp \mathsf{tt} \to \mathsf{c}_0\ .
\end{array}
$$

Figure 2: Dependency tuples of TRS $\mathcal{R}_{\mathsf{K}}$.

2. If $(\succsim, \succ)$ denotes a safe reduction pair, then it is also a $\mathcal{P}$-monotone complexity pair.

Proof. We consider the first assertion. For a proof by contradiction, suppose μ_{COM} is not a usable replacement map for $\mathcal{R}^\sharp$ in $\mathcal{P}^\sharp$. Thus there exists $s \in \to^*_{\mathcal{P}^\sharp}(\mathcal{T})$ and position $p \in \mathcal{P}\text{os}(s)$ such that $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}^\sharp, p} t$ for some term t, but $p \notin \mathcal{P}\text{os}_{\mu_{\text{COM}}}(s)$. Since $s \in \mathcal{T}^\sharp_\to$ by Lemma 3, symbols above position p in s are compound symbols, and so $p \notin \mathcal{P}\text{os}_\mu(s)$ by definition of μ_{COM}. This contradicts however that μ is a usable replacement map for $\mathcal{R}^\sharp$ in $\mathcal{P}^\sharp$.

From the first assertion one derives that reduction pairs are $\mathcal{P}$-monotone, and thus the second assertion holds. □

4.4. Usable Rules

In termination analysis it is standard to consider only those rewrite rules that can occur between applications of dependency pairs, the *usable rules*. Hirowaka and Moser [5] have shown that this technique can be safely employed for complexity analysis. The following definition captures an approximation of usable rules that looks at defined function symbols. Although this transformation is usually only of use on dependency pair problems, we nevertheless formulate it for the general case of complexity problems.

Definition 12 (Usable Rules)**.** Consider a complexity problem $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$.

1. Let $\mathcal{D}_\mathcal{P}$ collect the defined symbols in $\mathcal{P}$, i.e., $\mathcal{D}_\mathcal{P} := \{f \mid f(l_1, \dots, l_k) \to r \in \mathcal{P}\}$. We define the binary relation $\rhd_\text{d}$ on $\mathcal{D}_\mathcal{P}$ such that $f \rhd_\text{d} g$ holds if there exists a rule or dependency pair $f(l_1, \dots, l_k) \to r \in \mathcal{P}$ such that $g \in \mathcal{D}_\mathcal{P}$ occurs in r. We say that f *depends on* g.

2. The set of *usable symbols* $\mathcal{D}^\mathcal{U}_\mathcal{P}(t) \subseteq \mathcal{D}_\mathcal{P}$ *of a term* t is defined as

$$\mathcal{D}^\mathcal{U}_\mathcal{P}(t) := \{g \mid t \text{ contains symbol } f \ \in \mathcal{D}_\mathcal{P} \text{ with } f \rhd^*_\text{d} g\} .$$

   The notion of usable symbols is extended to sets of terms $\mathcal{T}$ by $\mathcal{D}^\mathcal{U}_\mathcal{P}(\mathcal{T}) := \bigcup_{t \in \mathcal{T}} \mathcal{D}^\mathcal{U}_\mathcal{P}(t)$.

3. The *usable rules* $\mathcal{U}_\mathcal{P}(\mathcal{R})$ in $\mathcal{P}$ of $\mathcal{R}$ are given by

$$\mathcal{U}_\mathcal{P}(\mathcal{R}) := \{f(l_1, \dots, l_k) \to r \in \mathcal{R} \mid f \in \mathcal{D}^\mathcal{U}_\mathcal{P}(\mathcal{T})\} .$$

The following auxiliary lemma shows that the usable symbols of starting terms $\mathcal{T}$ are closed under $\mathcal{P}$ reductions. In particular, this implies that only usable rules are ever triggered in derivations starting from $t \in \mathcal{T}$. Observe that the lemma crucially employs that starting terms are basic. This drastically simplifies the proof of soundness, compared to the setting of termination analysis [38].

Lemma 7. *Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$ denote a complexity problem. Then*

1. *$s \to_\mathcal{P} t$ implies $\mathcal{D}^\mathcal{U}_\mathcal{P}(t) \subseteq \mathcal{D}^\mathcal{U}_\mathcal{P}(s)$; and*
2. *$\mathcal{D}^\mathcal{U}_\mathcal{P}(\to^*_\mathcal{P}(\mathcal{T})) = \mathcal{D}^\mathcal{U}_\mathcal{P}(\mathcal{T})$.*

Proof. For the first assertion, suppose $s \to_\mathcal{P} t$ holds. Let $g \in \mathcal{D}^\mathcal{U}_\mathcal{P}(t)$, we show $g \in \mathcal{D}^\mathcal{U}_\mathcal{P}(s)$. Hence by assumption there exists a symbol $h \in \mathcal{D}_\mathcal{P}$ in t with $h \rhd^*_\text{d} g$. If h occurs also in s then by definition $g \in \mathcal{D}^\mathcal{U}_\mathcal{P}(s)$ holds, hence suppose h does not occur in s. As $s \to_\mathcal{P} t$, there exist a rule $f(l_1, \dots, l_k) \to r \in \mathcal{P}$, substitution σ and context C such that $s = C[f(l_1\sigma, \dots, l_k\sigma)]$ and $t = C[r\sigma]$. Since h does not occur in s but in t, it has to occurs in r. Hence $f \rhd_\text{d} h$ and thus $f \rhd^*_\text{d} g$. We obtain again $g \in \mathcal{D}^\mathcal{U}_\mathcal{P}(s)$ by definition of $\mathcal{D}^\mathcal{U}_\mathcal{P}(s)$ as desired.

Finally, the second assertion is a straightforward consequence of the first. □

Theorem 7 (Usable Rules Processor)**.** *Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$ denote a complexity problem. The following processor is sound and complete:*

$$\frac{\vdash \langle \mathcal{U}_\mathcal{P}(\mathcal{S})/\mathcal{U}_\mathcal{P}(\mathcal{W}), \mathcal{Q}, \mathcal{T} \rangle \colon f}{\vdash \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle \colon f} \ \textsf{Usable Rules.}$$

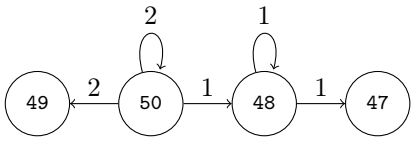


Figure 3: Dependency Graph of $\mathcal{P}^\sharp_{\mathsf{x\text{-}i}}$.

Proof. Let $\mathcal{R} \in \{\mathcal{S}, \mathcal{W}\}$ and $s \in \to^*_{\mathcal{P}}(\mathcal{T})$. Consider a step $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t$, where the rule $f(l_1, \dots, l_k) \to r \in \mathcal{R}$ has been applied. By definition, $f \in \mathcal{D}^{\mathcal{U}}_{\mathcal{P}}(s)$ and thus $f \in \mathcal{D}^{\mathcal{U}}_{\mathcal{P}}(\mathcal{T})$ as a consequence of Lemma 7(2). Hence $f(l_1, \dots, l_k) \to r \in \mathcal{U}_{\mathcal{P}}(\mathcal{R})$.

From this we obtain that every reduction $t_0 \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} t_1 \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} t_2 \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} \cdots$ for $t_0 \in \mathcal{T}$ turns into a reduction $t_0 \xrightarrow{\mathcal{Q}}_{\mathcal{U}_{\mathcal{P}}(\mathcal{S})/\mathcal{U}_{\mathcal{P}}(\mathcal{W})} t_1 \xrightarrow{\mathcal{Q}}_{\mathcal{U}_{\mathcal{P}}(\mathcal{S})/\mathcal{U}_{\mathcal{P}}(\mathcal{W})} t_2 \xrightarrow{\mathcal{Q}}_{\mathcal{U}_{\mathcal{P}}(\mathcal{S})/\mathcal{U}_{\mathcal{P}}(\mathcal{W})} \cdots$. Soundness of the processor follows. As the inverse direction trivially holds, we conclude completeness. □

Example 13 (Continued from Example 12). Consider the dependency pair problem $\langle \mathcal{S}^\sharp_{\mathsf{K}}/\mathcal{R}_{\mathsf{K}}, \mathcal{R}_{\mathsf{K}}, \mathcal{T}^\sharp_{\mathsf{K}}\rangle$ given in Example 12 with dependency pairs depicted in Figure 2. The defined symbols forest, kruskal and kruskal? are not usable with respect to this problem, the only usable rules are thus $\mathcal{U}_{\mathsf{K}} := \{9, 10, 15\text{–}46\}$. Application of the usable rules processor gives the following inference.

$$\frac{\vdash \langle \mathcal{S}^\sharp_{\mathsf{K}}/\mathcal{U}_{\mathsf{K}}, \mathcal{R}_{\mathsf{K}}, \mathcal{T}^\sharp_{\mathsf{K}}\rangle : n^2}{\vdash \langle \mathcal{S}^\sharp_{\mathsf{K}}/\mathcal{R}_{\mathsf{K}}, \mathcal{R}_{\mathsf{K}}, \mathcal{T}^\sharp_{\mathsf{K}}\rangle : n^2}\ \textsf{Usable Rules}.$$

Hence in total, the six rules $\{8, 11\text{–}14\}$ defining forest, kruskal and kruskal? are dropped from the weak component.

4.5. Dependency Graph Processors

The notion of *dependency graph*, a form of call and data flow graph, was initially proposed for the termination analysis [36]. We adapt this notion to complexity problems, compare also [15].

Definition 13 (Dependency Graph)**.** Let $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp \cup \mathcal{S}/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$ denote a DP problem.

1. The nodes of the *dependency graph* (*DG* for short) $\mathcal{G}$ of $\mathcal{P}^\sharp$ are the dependency pairs from $\mathcal{S}^\sharp \cup \mathcal{W}^\sharp$, and there is an arrow labeled by $i \in \mathbb{N}$ from $s^\sharp \to \textsc{com}(t^\sharp_1, \dots, t^\sharp_n)$ to $u^\sharp \to \textsc{com}(v^\sharp_1, \dots, v^\sharp_m)$ if for some substitutions $\sigma, \tau : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$, $t^\sharp_i\sigma \xrightarrow{\mathcal{Q}}^*_{\mathcal{S}\cup\mathcal{W}} u^\sharp\tau$ holds.
2. A graph $\mathcal{G}$ is called an *approximated dependency graph* for $\mathcal{P}^\sharp$ if it includes the dependency graph of $\mathcal{P}^\sharp$ in the following sense. The nodes of $\mathcal{G}$ are the dependency pairs from $\mathcal{S}^\sharp \cup \mathcal{W}^\sharp$, and whenever there is an arrow labeled by $i \in \mathbb{N}$ from $s^\sharp \to \textsc{com}(t^\sharp_1, \dots, t^\sharp_n)$ to $u^\sharp \to \textsc{com}(v^\sharp_1, \dots, v^\sharp_m)$ in the dependency graph of $\mathcal{P}^\sharp$, then this arrow occurs also in $\mathcal{G}$.

The dependency graph is not computable in general, however it is well understood how good approximations can be obtained [32].

Example 14 (Continued from Example 7). Figure 3 depicts the dependency graph of the DP problem $\mathcal{P}^\sharp_{\mathsf{x\text{-}i}} = \langle \mathcal{S}^\sharp_{\mathsf{x}}/\mathcal{R}_{\mathsf{x}}, \mathcal{R}_{\mathsf{x}}, \mathcal{T}^\sharp_{\mathsf{x}}\rangle$ with $\mathcal{S}^\sharp_{\mathsf{x}} = \{47\text{–}50\}$ and $\mathcal{R}_{\mathsf{x}}$ depicted in Example 7 and Example 3, respectively.

In the following, we introduce various processors that transform a DP problem $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp \cup \mathcal{S}/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$, based on an analysis of the (approximated) dependency graph of $\mathcal{P}^\sharp$. To avoid reasoning up to permutations of rewrite steps, we introduce a notion of *derivation tree* that disregards the order of parallel steps under compound contexts. This notion rest on the observation that any term $t \in \to_{\mathcal{P}^\sharp}(\mathcal{T}^\sharp)$ is of the form $t = C[t_1, \dots, t_n]$ for a maximal compound context C by Lemma 3. Thus any further reduction of t consists of possibly interleaved, but otherwise *independent*, reductions of the terms $t_1, \dots, t_n$.

Derivation trees are *hypergraphs* where nodes are labeled by terms and edges by rules. Here, a (directed) *hypergraph* over *labels* $\mathcal{L}$ is a triple $G = (N, E, \mathsf{lab})$ where N is a set of *nodes*, $E \subseteq N \times \mathcal{P}(N)$ a set of *edges*, and $\mathsf{lab} : N \cup E \to \mathcal{L}$ a *labeling function*. For $e = \langle u, \{v_1, \dots, v_n\}\rangle \in E$ we call the node u the *source*, and nodes $v_1, \dots, v_n$ the *targets* of e. We denote by $\rightharpoonup_G$ the *successor relation* in the hypergraph G, i.e. $u \rightharpoonup_G v$ if there exists an edge $e = \langle u, \{v_1, \dots, v_n\}\rangle \in E$ with $v \in \{v_1, \dots, v_n\}$. We set $u \xrightarrow[G]{\mathcal{K}} v$ for labels $\mathcal{K} \subseteq \mathcal{L}$ if additionally $\mathsf{lab}(e) \in \mathcal{K}$ holds, and abbreviate $\xrightarrow[G]{\{l\}}$ by $\xrightarrow[G]{l}$. If there exists a *path* $u = w_1 \rightharpoonup_G \cdots \rightharpoonup_G w_n = v$ we say that v is *reachable* from u in G. We call a hypergraph G a *hypertree* (*tree* for short) if there exists a unique node $u \in N$, the *root* of G, such that every $v \in N$ is reachable from u by a unique path. We keep the convention that every node is the source of at most one edge.

Definition 14 (Derivation Tree)**.** Let $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp \cup \mathcal{S}/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$ denote a dependency pair problem.

1. Consider a term $t \in \mathcal{T}^\sharp(\mathcal{F}, \mathcal{V}) \cup \mathcal{T}(\mathcal{F}, \mathcal{V})$. The set of $\mathcal{P}^\sharp$ *derivation trees of* t is defined as the least set of hypertrees such that:
   (a) T is a $\mathcal{P}^\sharp$ derivation tree of t where T consists of a unique node labeled by t.
   (b) Suppose $t \xrightarrow[\{l\to r\}]{\mathcal{Q}} \textsc{com}(t_1, \dots, t_n)$ for $l \to r \in \mathcal{P}^\sharp$ and let T_i denote a $\mathcal{P}^\sharp$ derivation tree of t_i for $i = 1, \dots, n$. Then T is a $\mathcal{P}^\sharp$ derivation tree of t, where T is a tree with T_i $(i = 1, \dots, n)$ rooted at its i^{th} child, the root of T is labeled by t, and the edge from the root of T to its children is labeled by $l \to r$.
2. For a $\mathcal{P}^\sharp$ derivation tree T we denote by $|T|_{\mathcal{R}^\sharp \cup \mathcal{R}}$ the number of edges labeled by a rule or dependency pair $l \to r \in \mathcal{R}^\sharp \cup \mathcal{R}$.

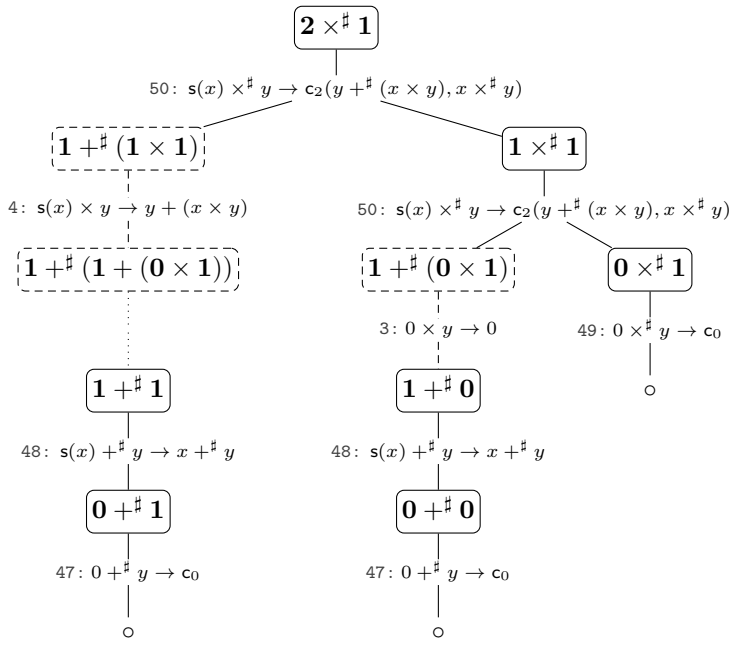


Figure 4: $\mathcal{P}^\sharp_{\text{×-i}}$ derivation tree of $\mathbf{2} \times^\sharp \mathbf{1}$.

Consider a $\mathcal{P}^\sharp$ derivation tree T. Every edge $\langle u, \{v_1, \dots, v_n\}\rangle$ in T *corresponds* to a rewrite steps $t \xrightarrow[l\to r]{\mathcal{Q}} \textsc{com}(t_1, \dots, t_n)$, with t and $t_1, \dots, t_n$ precisely the label of source u and targets $v_1, \dots, v_n$ respectively, and $l \to r$ the label of the considered edge. We also say that $l \to r$ was *applied* at node u in T. This correspondence can be lifted to rewrite sequences, and motivates our notion of the size $|T|_{\mathcal{R}^\sharp \cup \mathcal{R}}$ of T with respect to $\mathcal{R}^\sharp \cup \mathcal{R}$: $|T|_{\mathcal{R}^\sharp \cup \mathcal{R}}$ refers to the number applications of rules from $\mathcal{R}^\sharp \cup \mathcal{R}$.

Example 15 (Continued from Example 7). Recall the DP problem $\mathcal{P}^\sharp_{\times\text{-i}} = \langle \mathcal{S}^\sharp_\times / \mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}^\sharp_\times \rangle$ from Example 7. In Figure 4 we depict a $\mathcal{P}^\sharp_{\times\text{-i}}$ derivation tree $T_\times$ of the term $\mathbf{2} \times^\sharp \mathbf{1}$. Solid nodes indicate applications of DPs from the strict component $\mathcal{S}^\sharp_\times$. Conversely, dashed nodes indicate applications of weak rules $\mathcal{R}_\times$. The dotted lines indicate that we left out some rewrite steps with respect to $\mathcal{R}_\times$. We have $|T_\times|_{\mathcal{S}^\sharp_\times} = 7$.

Lemma 8. *Let $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp \cup \mathcal{S} / \mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$ be a DP problem. Then for every $t \in \mathcal{T}(\mathcal{F}, \mathcal{V}) \cup \mathcal{T}^\sharp(\mathcal{F}, \mathcal{V})$, we have*

$$\mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{S}^\sharp \cup \mathcal{S} / \mathcal{W}^\sharp \cup \mathcal{W}}) =_\mathsf{k} \max\{|T|_{\mathcal{S}^\sharp \cup \mathcal{S}} \mid T \text{ is a } \mathcal{P}^\sharp\text{-derivation tree of } t\} \,.$$

In particular,

$$\mathsf{cp}_{\mathcal{P}^\sharp}(n) =_\mathsf{k} \max\{|T|_{\mathcal{S}^\sharp \cup \mathcal{S}} \mid T \text{ is a } \mathcal{P}^\sharp\text{-derivation tree of } t \in \mathcal{T}^\sharp \text{ with } |t| \leqslant n\} \,,$$

holds.

Proof. We consider the first assertion. Let $t \in \mathcal{T}(\mathcal{F}, \mathcal{V}) \cup \mathcal{T}^\sharp(\mathcal{F}, \mathcal{V})$, and abbreviate

$$s := \max\{|T|_{\mathcal{S}^\sharp \cup \mathcal{S}} \mid T \text{ is a } \mathcal{P}^\sharp\text{-derivation tree of } t\} \qquad \text{and} \qquad \ell := \mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{S}^\sharp \cup \mathcal{S} / \mathcal{W}^\sharp \cup \mathcal{W}}) \,.$$

We show $\ell \geqslant_\mathsf{k} s$ and $s \geqslant_\mathsf{k} \ell$. For the first inequality, suppose s is well-defined. Hence there exists a $\mathcal{P}^\sharp$ derivation tree T of t with $|T|_{\mathcal{S}^\sharp \cup \mathcal{S}} = s$. A breadth first traversal on T constructs a $\mathcal{P}^\sharp$ derivation D, such that every application of $l \to r \in \mathcal{S}^\sharp \cup \mathcal{S}$ translates to an application of $l \to r$ in D. This derivation D then witnesses $\ell \geqslant_\mathsf{k} s$.

Inversely, for the second equality one can construct for an arbitrary $\mathcal{P}^\sharp$ derivation D starting from $t \in \mathcal{T}^\sharp$ a $\mathcal{P}^\sharp$ derivation tree T. Every application of a rule $l \to r \in \mathcal{P}^\sharp$ in D translates to a unique edge in T. The construction is carried out by induction on the length of D. One uses that the final term in this derivation is of the form $C[u_1, \dots, u_n] \in \mathcal{T}^\sharp_\to$ for C a maximal compound context (compare Lemma 3). Note also that for each sub-term u_i $(i = 1, \dots, n)$, the constructed tree T contains a dedicated leaf labeled by u_i. This shows $s \geqslant_\mathsf{k} \ell$. □

Remark. Our notion of derivation trees is related to the notion of *chain tree* given by Noschinski et al. [34]. Whereas in $\mathcal{P}^\sharp$ derivation trees each edge corresponds to a single rewrite step, in chain trees an edge corresponds to a dependency pair step $l^\sharp\sigma \to \textsc{com}(r^\sharp_1\sigma, \dots, r^\sharp_k\sigma)$ followed by normalisation of unmarked sub-terms in the reduct. This notion is of limited use in our setting, as we also want to account for the normalisation steps in Lemma 8.

Note that the dependency graph $\mathcal{G}$ of $\mathcal{P}^\sharp$ indicates in which order dependency pairs can occur in a derivations of $\mathcal{P}^\sharp$. To make this intuition precise, we adapt the notion of *DP chain* known from termination analysis. Recall that for a derivation tree T, the symbol $\rightharpoonup_T$ denotes the child relation, and $\xrightarrow{\mathcal{R}}_T$ its restriction to edges labeled by $l \to r \in \mathcal{R}$.

Definition 15 (Dependency Pair Chain)**.** Let T be a derivation tree, and consider a path

$$u_1 \xrightarrow{\{l_1 \to r_1\}}_T \cdot \xrightarrow{\mathcal{S} \cup \mathcal{W}}{}^*_T u_2 \xrightarrow{\{l_2 \to r_2\}}_T \cdot \xrightarrow{\mathcal{S} \cup \mathcal{W}}{}^*_T \cdots \,,$$

for a sequence of dependency pairs $C \colon l_1 \to r_1, l_2 \to r_2, \dots$. The sequence C is called a *dependency pair chain (in T)*, or *DP chain* for brevity.

Example 16 (Continued from Example 14). Reconsider the $\mathcal{P}^\sharp_{\times\text{-i}}$ derivation tree $T_\times$ given in Figure 4. This tree gives rise to three maximal chains: the chain 50,48,47 along the left; the chain 50,50,48,47 along the middle and the chain 50,50,49 along the right path.

Lemma 9. *Every chain in a $\mathcal{P}^\sharp$ derivation tree is a path in the dependency graph of the DP problem $\mathcal{P}^\sharp$.*

Proof. Let $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp \cup \mathcal{S}/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$ and consider two successive DPs $l_1 \to r_1 := s^\sharp \to \textsc{com}(t_1^\sharp, \dots, t_n^\sharp)$ and $l_2 \to r_2 := u^\sharp \to \mathsf{c}_m(v_1^\sharp, \dots, v_m^\sharp)$ in a DP chain of a $\mathcal{P}^\sharp$ derivation tree T. Thus there exists nodes u_1, u_2, v_1, v_2 with

$$u_1 \xrightarrow{l_1 \to r_1\searrow}_T u_2 \xrightarrow{\mathcal{S}\cup\mathcal{W}\searrow}{}^*_T v_1 \xrightarrow{l_2 \to r_2\searrow}_T v_2 \,,$$

and thus there exist substitutions σ, τ such that u_2 is labeled by $t_i^\sharp\sigma$ for some $i \in \{1, \dots, n\}$ and v_1 by $u^\sharp\tau$. As $u_2 \xrightarrow{\mathcal{S}\cup\mathcal{W}\searrow}{}^*_T v_1$ we have $t_i^\sharp\sigma \xrightarrow{\mathcal{Q}}{}^*_{\mathcal{S}\cup\mathcal{W}} u^\sharp\tau$ by definition, and thus there is an edge from $l_1 \to r_1$ to $l_2 \to r_2$ in the DG of $\mathcal{P}^\sharp$, and hence in $\mathcal{G}$. The lemma follows from this. □

4.5.1. Simplification Techniques

In this section we introduce a handful of syntactic simplifications. None of the observations are very deep. Nevertheless the resulting processors are important. They allow the reduction of a complexity problem to a core set of rules reflecting the complexity of the input problem asymptotically.

Remove Weak Suffixes Processor. The *leaf removal* processor introduced in [34] states that all dependency pairs that occur as *leafs* in the DG, that is, dependency pairs that constitute nodes in the DG without outgoing edges, can be dropped from the input problem. This processor is sound in the setting of [34] where all dependency pairs occur in the strict component. Without further restrictions, this processor is unsound in our setting.

Example 17. The following inference is not sound

$$\frac{\vdash \langle \varnothing/\{\mathsf{f}^\sharp \to \mathsf{c}_2(\mathsf{f}^\sharp, \mathsf{g}^\sharp)\}, \varnothing, \{\mathsf{f}^\sharp\}\rangle\colon f}{\vdash \langle \{\mathsf{g}^\sharp \to \mathsf{c}_0\}/\{\mathsf{f}^\sharp \to \mathsf{c}_2(\mathsf{f}^\sharp, \mathsf{g}^\sharp)\}, \varnothing, \{\mathsf{f}^\sharp\}\rangle\colon f} \,,$$

despite the fact that $\mathsf{g}^\sharp \to \mathsf{c}_0$ is a leaf in the dependency graph of the input problem. Observe that the complexity function of the input problem is undefined for inputs of sizes greater than two, whereas the generated problem has trivially constant complexity.

Another situation where removal of leafs leads to problems is when our analysis has to account for ordinary rewrite rules beside dependency pairs. Again this case is a priori excluded in [34].

Example 18. Consider the following inference, where $\mathsf{f}^\sharp \to \mathsf{g}^\sharp(\mathsf{h})$ denotes a leaf in the dependency graph of the input problem.

$$\frac{\vdash \langle \{\mathsf{h} \to \mathsf{h}\}/\varnothing, \varnothing, \{\mathsf{f}^\sharp\}\rangle\colon f}{\vdash \langle \{\mathsf{f}^\sharp \to \mathsf{g}^\sharp(\mathsf{h}),\ \mathsf{h} \to \mathsf{h}\}/\varnothing, \varnothing, \{\mathsf{f}^\sharp\}\rangle\colon f} \,,$$

Then the generated problem has constant complexity, whereas the complexity function of the input problem is undefined for inputs greater than two.

We can however remove dependency pairs from the weak component that do not trigger rewrite steps with respect to the strict component. Provided that the strict component of the input problem constitutes of dependency pairs only, rules amenable for removal can be determined based on the dependency graph.

Definition 16 (Forward Closed)**.** Let $\mathcal{G}$ be a dependency graph and let $\mathcal{R}^\sharp$ denote a set of dependency pairs. We say that $\mathcal{R}^\sharp$ is *closed under $\mathcal{G}$-successors* if for every edge from $s \to t \in \mathcal{R}^\sharp$ to $u \to v$ in $\mathcal{G}$ we have that also $u \to v \in \mathcal{R}^\sharp$. For a complexity problem $\mathcal{P}^\sharp$, we call a set of DPs occurring in $\mathcal{P}^\sharp$ *forward closed* if this set is closed under $\mathcal{G}$-successors for the dependency graph $\mathcal{G}$ of $\mathcal{P}^\sharp$.

Note that forward closure of a set of DPs can be approximated based on an estimated dependency graph.

Theorem 8 (Remove Weak Suffixes Processor)**.** *Fix a DP problem $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp/\mathcal{W}^\sharp \cup \mathcal{W}_l^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$ where $\mathcal{W}_l^\sharp$ is forward closed and $\mathcal{S}^\sharp \cap \mathcal{W}_l^\sharp = \varnothing$. The following processor is sound and complete.*

$$\frac{\vdash \langle \mathcal{S}^\sharp/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle\colon f}{\vdash \langle \mathcal{S}^\sharp/\mathcal{W}^\sharp \cup \mathcal{W}_l^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle\colon f}\ \textsf{Remove Weak Suffixes}\,.$$

Proof. Let $\mathcal{P}^\sharp_g := \langle \mathcal{S}^\sharp / \mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$. The processor is trivially complete. To see that the processor is sound, we show that for every $\mathcal{P}^\sharp$ derivation tree T of $t^\sharp \in \mathcal{T}^\sharp$, there exists a $\mathcal{P}^\sharp_g$ derivation tree T of $t^\sharp$ with $|T_g|_{\mathcal{S}^\sharp} = |T|_{\mathcal{S}^\sharp}$. Hence soundness follows by Lemma 8.

Consider a $\mathcal{P}^\sharp$ derivation tree T of $t^\sharp$. We define T_g as the derivation tree of $t^\sharp$ that is obtained by removing applications of dependency pairs from $\mathcal{W}^\sharp_l$. More precise, whenever there is an edge e labeled with a DP from $\mathcal{W}^\sharp_l$ in T, we remove e and the sub-trees rooted in the target nodes of e. Then, by construction, T_g is a $\mathcal{P}^\sharp_g$ derivation tree of $t^\sharp$. Suppose now $|T|_{\mathcal{S}^\sharp} \neq |T_g|_{\mathcal{S}^\sharp}$, i.e. $|T|_{\mathcal{S}^\sharp} > |T_g|_{\mathcal{S}^\sharp}$. Then T_g misses a sub-tree of T with an edge labeled by a DP $l \to r \in \mathcal{S}^\sharp$. This assumption gives rise to a DP chain where $l \to r$ is preceded by a DP from $\mathcal{W}^\sharp_l$. Since $\mathcal{S}^\sharp \cap \mathcal{W}^\sharp_l = \varnothing$ by assumption, the DP $l \to r$ does not occur in $\mathcal{W}^\sharp_l$, hence the DP chain contradicts that $\mathcal{W}^\sharp_l$ is forward closed, by Lemma 9. □

Predecessor Estimation. Noschinski et al. [34] observed that the application of a dependency pair $l \to r$ in a $\mathcal{P}^\sharp$ derivation can be estimated in terms of the application of its predecessors in the dependency graph of $\mathcal{P}^\sharp$.

Definition 17. Let $\mathcal{G}$ be an approximated dependency graph. We denote by $\mathsf{Pre}_\mathcal{G}(l \to r)$ the set of all (direct) *predecessors* of node $l \to r$ in $\mathcal{G}$. For a set of dependency pairs $\mathcal{R}^\sharp$ we set

$$\mathsf{Pre}_\mathcal{G}(\mathcal{R}^\sharp) := \bigcup_{l \to r \in \mathcal{R}^\sharp} \mathsf{Pre}_\mathcal{G}(l \to r) \ .$$

Then for an approximated dependency graph $\mathcal{G}$ of $\mathcal{P}^\sharp$, we have the following correspondence between the number of applications of dependency pairs from $\mathcal{R}^\sharp$ and $\mathsf{Pre}_\mathcal{G}(\mathcal{R}^\sharp)$.

Lemma 10. *Let $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp \cup \mathcal{S} / \mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$ denote a DP problem, let $\mathcal{G}$ be an approximated dependency graph for $\mathcal{P}^\sharp$, and let $l \to r \in \mathcal{P}^\sharp$ denote a dependency pair. For every $\mathcal{P}^\sharp$-derivation tree T,*

$$|T|_{\mathcal{R}^\sharp \cup \mathcal{R}} \leqslant \max\{1, |T|_{(\mathcal{R}^\sharp \setminus \{l \to r\}) \cup \mathsf{Pre}_\mathcal{G}(l \to r) \cup \mathcal{R}} \cdot \Delta\} \ ,$$

where Δ denote the maximal arity of a compound symbol occurring in a rule from $\mathcal{P}^\sharp$.

Proof. Consider the non-trivial case $l \to r \notin \mathsf{Pre}_\mathcal{G}(l \to r)$ and let T denote a $\mathcal{P}^\sharp$ derivation tree with an edge labeled by $l \to r \in \mathcal{R}^\sharp$. To conclude the lemma, we bind the applications of the DP $l \to r$ in T terms of applications of DPs $\mathsf{Pre}_\mathcal{G}(l \to r)$. More precise, we show $|T|_{\{l \to r\}} \leqslant \max\{1, |T|_{\mathsf{Pre}_\mathcal{G}(l \to r)} \cdot \Delta\}$. By Lemma 9 chains of T translate to paths in $\mathcal{G}$. Hence if $l \to r$ occurs in a DP chain of T, it is either the first DP in the DP chain, or is headed by a DP from $\mathsf{Pre}_\mathcal{G}(l \to r)$. In the former case $|T|_{\{l \to r\}} = 1$. In the latter case, let $\{u_1, \ldots, u_n\}$ collect all sources of $l \to r$ edges in T. To each node $u_i \in \{u_1, \ldots, u_n\}$ we can identify a unique node $\mathsf{pre}(u_i)$ such that $\mathsf{pre}(u_i) \xrightarrow{\mathsf{Pre}_\mathcal{G}(l \to r)\cup}_T \cdot \xrightarrow{\mathcal{S} \cup \mathcal{W}\cup}{}^*_T u_i$ holds. Let $\{v_1, \ldots, v_m\} = \{\mathsf{pre}(u_1), \ldots, \mathsf{pre}(u_n)\}$. Since $\xrightarrow{\mathcal{S} \cup \mathcal{W}\cup}_T$ is non-branching, and $\mathsf{pre}(u_i)$ has at most Δ successors, it follows that $|T|_{\{l \to r\}} = n \leqslant \Delta \cdot m \leqslant \Delta \cdot |T|_{\mathsf{Pre}_\mathcal{G}(l \to r)}$. □

Theorem 9 (Predecessor Estimation Processor)**.** *Let $\mathcal{G}$ denote an approximated dependency graph of the DP problem $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp_1 \cup \mathcal{S}^\sharp_2 \cup \mathcal{S} / \mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$. The following processor is sound:*

$$\frac{\vdash \langle \mathsf{Pre}_\mathcal{G}(\mathcal{S}^\sharp_1) \cup \mathcal{S}^\sharp_2 \cup \mathcal{S} / \mathcal{S}^\sharp_1 \cup \mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle \colon f}{\vdash \langle \mathcal{S}^\sharp_1 \cup \mathcal{S}^\sharp_2 \cup \mathcal{S} / \mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle \colon f} \ \mathsf{Predecessor\ Estimation} \ .$$

Proof. Soundness follows trivially from Lemma 10, employing Lemma 8. □

We point out that the predecessor estimation processor is an adaptation of *knowledge propagation* introduced in [34], not relying on a dedicated component $\mathcal{K}$ of known rules.

Application of the predecessor estimation processor makes only sense if $\mathsf{Pre}_\mathcal{G}(\mathcal{S}^\sharp_1) \neq \mathcal{S}^\sharp_1$. On the other hand, it is always safe to take for $\mathcal{S}^\sharp_1$ a maximal set of DPs such that $\mathsf{Pre}_\mathcal{G}(\mathcal{S}^\sharp_1) \subseteq \mathcal{S}^\sharp_2$.

The combination of Theorem 8 and Theorem 9 allows us to remove DPs $\mathcal{R}^\sharp$ that occur as leafs in the DG $\mathcal{G}$ of the input problem, provided $\mathsf{Pre}_\mathcal{G}(\mathcal{R}^\sharp)$ constitutes of dependency pairs that occur in the strict component, as in [34]. This is clarified on our running example.

Example 19 (Continued from Example 13). Observe that the DPs

$$\mathcal{L}^\sharp := \{61\text{–}63, 65, 67, 71, 73, 77, 79, 81, 83, 85, 87\text{–}89, 91\text{–}93, 95\text{–}102\}$$

depicted in Example 12 occur as leafs in the dependency graph $\mathcal{G}$ of the complexity problem $\langle \mathcal{S}^\sharp_\mathsf{K}/\mathcal{U}_\mathsf{K}, \mathcal{R}_\mathsf{K}, \mathcal{T}^\sharp_\mathsf{K}\rangle$. Instantiating $\mathcal{S}^\sharp_1$ by $\mathcal{L}^\sharp$ and $\mathcal{S}^\sharp_2$ by

$$\mathcal{S}^\sharp_\mathsf{K} \setminus \mathcal{L}^\sharp := \{64, 66, 68\text{–}70, 72, 74\text{–}76, 78, 80, 82, 84, 86, 90, 94\} \,,$$

in Theorem 9, we can continue the complexity proof of Example 13 as follows.

$$\dfrac{\dfrac{\vdash \langle (\mathcal{S}^\sharp_\mathsf{K} \setminus \mathcal{L}^\sharp)/\mathcal{U}_\mathsf{K}, \mathcal{R}_\mathsf{K}, \mathcal{T}^\sharp_\mathsf{K}\rangle \colon n^2}{\vdash \langle (\mathcal{S}^\sharp_\mathsf{K} \setminus \mathcal{L}^\sharp)/\mathcal{L}^\sharp \cup \mathcal{U}_\mathsf{K}, \mathcal{R}_\mathsf{K}, \mathcal{T}^\sharp_\mathsf{K}\rangle \colon n^2}\,\textsf{Remove Weak Suffixes}}{\vdash \langle \mathcal{S}^\sharp_\mathsf{K}/\mathcal{U}_\mathsf{K}, \mathcal{R}_\mathsf{K}, \mathcal{T}^\sharp_\mathsf{K}\rangle \colon n^2}\,\textsf{Predecessor Estimation} \,.$$

Observe that we employ $\mathsf{Pre}_\mathcal{G}(\mathcal{L}^\sharp) \subseteq \mathcal{S}^\sharp_\mathsf{K} \setminus \mathcal{L}^\sharp$, and that $\mathcal{L}^\sharp$ is trivially forward closed in the intermediate problem.

Simplifying Right-hand Sides. In the proof step given in Example 19 we have removed all dependency pairs defining the dependency pair symbols $\mathsf{src}^\sharp$, $\mathsf{wt}^\sharp$ and $\mathsf{trg}^\sharp$ as well as $\vee^\sharp$ and $\wedge^\sharp$. Since the strict component of the considered DP problem contains only dependency pairs, it is safe to remove any calls to these symbols in right-hand sides. For instance, the dependency pair

$$72\colon\quad \mathsf{inBlock}^\sharp(e, p :: P) \to \mathsf{c}_7\big((\mathsf{src}(e) \in p \wedge \mathsf{trg}(e) \in p) \vee^\sharp \mathsf{inBlock}(e, P), \mathsf{src}(e) \in p \wedge^\sharp \mathsf{trg}(e) \in p, \\ \mathsf{src}(e) \in^\sharp p, \mathsf{trg}(e) \in^\sharp p, \mathsf{src}^\sharp(e), \mathsf{trg}(e)^\sharp, \mathsf{inBlock}^\sharp(e, P)\big) \,,$$

can be simplified to

$$72\mathsf{s}\colon\quad \mathsf{inBlock}^\sharp(e, p :: P) \to \mathsf{c}_3(\mathsf{src}(e) \in^\sharp p, \mathsf{trg}(e) \in^\sharp p, \mathsf{inBlock}^\sharp(e, P)) \,.$$

Although this simplification has no effect on the complexity of the considered problem, it still makes an automated analysis often easier. For instance, as a result of the transformation the usable rules, and also constraints for complexity pairs, can get simpler. The next processor provides a formalisation of this idea, which was first implemented in AProVE [34].

Theorem 10 (Simplify Right-hand Sides Processor)**.** *Let $\mathcal{G}$ denote an approximated dependency graph of the DP problem $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$. For dependency pairs $l^\sharp \to \mathrm{COM}(r^\sharp_1, \ldots, r^\sharp_k) \in \mathcal{S}^\sharp \cup \mathcal{W}^\sharp$, call an argument $r^\sharp_i$ removable if there is no outgoing edge from $l^\sharp \to \mathrm{COM}(r^\sharp_1, \ldots, r^\sharp_k)$ labeled by $i \in \{1, \ldots, k\}$. Define*

$$\mathsf{simp}_\mathcal{G}(l^\sharp \to \mathrm{COM}(r^\sharp_1, \ldots, r^\sharp_k)) := l^\sharp \to \mathrm{COM}(r^\sharp_{i_1}, \ldots, r^\sharp_{i_l}) \,,$$

where $\{r^\sharp_{i_1}, \ldots, r^\sharp_{i_l}\} \subseteq \{r^\sharp_1, \ldots, r^\sharp_k\}$ collects all arguments of the right-hand side which are not removable. We denote by $\mathsf{simp}_\mathcal{G}$ also its homomorphic extensions to sets.

The following processor is sound and complete.

$$\dfrac{\vdash \langle \mathsf{simp}_\mathcal{G}(\mathcal{S}^\sharp)/\mathsf{simp}_\mathcal{G}(\mathcal{W}^\sharp) \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle \colon f}{\vdash \langle \mathcal{S}^\sharp/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle \colon f}\,\textsf{Simplify RHS} \,.$$

Proof. Let $\mathcal{P}^\sharp_g$ denote the generated problem $\langle \mathsf{simp}_\mathcal{G}(\mathcal{S}^\sharp)/\mathsf{simp}_\mathcal{G}(\mathcal{W}^\sharp) \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$.

Consider a $\mathcal{P}^\sharp$ derivation tree T of $t^\sharp \in \mathcal{T}^\sharp$. Call a sub-tree T_i resulting from the application of a DP $l^\sharp \to \mathrm{COM}(r^\sharp_1, \ldots, r^\sharp_k) \in \mathcal{S}^\sharp \cup \mathcal{W}^\sharp$ in T *removable*, if its root is labeled by a term $r^\sharp_i\sigma$ for some removable argument $r^\sharp_i$ and substitution σ. By Lemma 9 and the assumption that $r^\sharp_i$ is removable, T_i contains no

$$
\begin{aligned}
64: \quad \mathsf{forest}^\sharp(\mathsf{graph}(N,E)) &\to \mathsf{c}_3(\mathsf{kruskal}^\sharp(\mathsf{sort}(E),[\,],\mathsf{partitions}(N)),\mathsf{sort}^\sharp(E),\mathsf{partitions}^\sharp(N))\\
66: \quad \mathsf{partitions}^\sharp(n :: N) &\to \mathsf{partitions}^\sharp(N)\\
68: \quad \mathsf{kruskal}^\sharp(e :: E,W,P) &\to \mathsf{c}_2(\mathsf{kruskal?}^\sharp(\mathsf{inBlock}(e,P),e,E,W,P),\mathsf{inBlock}^\sharp(e,P))\\
69: \quad \mathsf{kruskal?}^\sharp(\mathsf{tt},e,E,W,P) &\to \mathsf{kruskal}^\sharp(E,W,P)\\
70: \quad \mathsf{kruskal?}^\sharp(\mathsf{ff},e,E,W,P) &\to \mathsf{c}_2(\mathsf{kruskal}^\sharp(E,e :: W,\mathsf{join}(e,P,[\,])),\mathsf{join}^\sharp(e,P,[\,]))\\
72s: \quad \mathsf{inBlock}^\sharp(e,p :: P) &\to \mathsf{c}_3(\mathsf{src}(e) \in^\sharp p,\mathsf{trg}(e) \in^\sharp p,\mathsf{inBlock}^\sharp(e,P))\\
74s: \quad \mathsf{join}^\sharp(e,p :: P,q) &\to \mathsf{c}_2(\mathsf{join?}^\sharp(\mathsf{src}(e) \in p \vee \mathsf{trg}(e) \in p,e,p,P,q),\mathsf{src}(e) \in^\sharp p)\\
75: \quad \mathsf{join?}^\sharp(\mathsf{tt},e,p,P,q) &\to \mathsf{c}_2(\mathsf{join}^\sharp(e,P,p +\!\!+ q),p +\!\!+^\sharp q)\\
76: \quad \mathsf{join?}^\sharp(\mathsf{ff},e,p,P,q) &\to \mathsf{join}^\sharp(e,P,q)\\
78: \quad \mathsf{sort}^\sharp(e :: E) &\to \mathsf{c}_2(\mathsf{insert}^\sharp(e,\mathsf{sort}(E)),\mathsf{sort}^\sharp(E))\\
80s: \quad \mathsf{insert}^\sharp(e,f :: E) &\to \mathsf{c}_2(\mathsf{insert?}^\sharp(\mathsf{wt}(e) \leqslant \mathsf{wt}(f),e,f,E),\mathsf{wt}(e) \leqslant^\sharp \mathsf{wt}(f))\\
82: \quad \mathsf{insert?}^\sharp(\mathsf{ff},e,f,E) &\to \mathsf{insert}^\sharp(e,E)\\
84s: \quad n \in^\sharp (m :: p) &\to \mathsf{c}_2(n =^\sharp m, n \in^\sharp p)\\
86: \quad (n :: p) +\!\!+^\sharp q &\to p +\!\!+^\sharp q\\
90: \quad \mathsf{s}(x) =^\sharp \mathsf{s}(y) &\to x =^\sharp y\\
94: \quad \mathsf{s}(x) \leqslant^\sharp \mathsf{s}(y) &\to x \leqslant^\sharp y\ .
\end{aligned}
$$

Figure 5: Simplified dependency pairs $\mathcal{S}^\sharp_{\mathsf{Ks}}$.

applications of a DP, i.e. $|T_i|_{\mathcal{S}^\sharp} = 0$. By deleting all removable sub-trees of T, and replacing the application of $l^\sharp \to \textsc{com}(r^\sharp_1,\ldots,r^\sharp_k) \in \mathcal{S}^\sharp \cup \mathcal{W}^\sharp$ by $\mathsf{simp}_\mathcal{G}(l^\sharp \to \textsc{com}(r^\sharp_1,\ldots,r^\sharp_k)) \in \mathsf{simp}_\mathcal{G}(\mathcal{S}^\sharp) \cup \mathsf{simp}_\mathcal{G}(\mathcal{W}^\sharp)$ we can thus construct a $\mathcal{P}^\sharp_g$ derivation tree T_g with $|T_g|_{\mathsf{simp}_\mathcal{G}(\mathcal{S}^\sharp)} = |T|_{\mathcal{S}^\sharp}$. This concludes soundness of the processor, by Lemma 8.

Inversely, completeness is shown by constructing from every $\mathcal{P}^\sharp_g$ derivation tree a $\mathcal{P}^\sharp$ derivation tree, obtained by replacing applications of $\mathsf{simp}_\mathcal{G}(l^\sharp \to \textsc{com}(r^\sharp_1,\ldots,r^\sharp_k))$ by applications of $l^\sharp \to \textsc{com}(r^\sharp_1,\ldots,r^\sharp_k)$.□

Example 20 (Continued from Example Example 19). By Theorem 10 the following inference is sound.

$$\frac{\vdash \langle \mathcal{S}^\sharp_{\mathsf{Ks}}/\mathcal{U}_\mathsf{K},\mathcal{R}_\mathsf{K},\mathcal{T}^\sharp_\mathsf{K}\rangle\colon n^2}{\vdash \langle (\mathcal{S}^\sharp_\mathsf{K} \setminus \mathcal{L}^\sharp)/\mathcal{U}_\mathsf{K},\mathcal{R}_\mathsf{K},\mathcal{T}^\sharp_\mathsf{K}\rangle\colon n^2}\ \mathsf{Simplify\ RHS}\ .$$

Here $\mathcal{S}^\sharp_{\mathsf{Ks}} := \mathsf{simp}_\mathcal{G}(\mathcal{S}^\sharp_\mathsf{K} \setminus \mathcal{L}^\sharp)$ consists of the dependency pairs depicted in Figure 5.

4.5.2. Dependency Graph Decomposition

The simplified DP problem $\langle \mathcal{S}^\sharp_{\mathsf{Ks}}/\mathcal{U}_\mathsf{K},\mathcal{R}_\mathsf{K},\mathcal{T}^\sharp_\mathsf{K}\rangle$ contains roughly half of the rules from the DP problem $\langle \mathcal{S}^\sharp_\mathsf{K}/\mathcal{R}_\mathsf{K},\mathcal{R}_\mathsf{K},\mathcal{T}^\sharp_\mathsf{K}\rangle$. Still, TCT is not able to synthesise orders that orient all of the remaining dependency pairs,[5] even in an iterated fashion using the relative decomposition processor from Theorem 4. Motivated by the inability to synthesise suitable orders for larger examples, we introduce a novel technique, called *dependency graph decomposition* (*DG decomposition* for short). The aim of this transformation technique

[5]This statement remains correct for comparable provers like AProVE and CaT.

is to decompose the input problem into several pieces that are manageable by complexity pairs. Our work on this processor is also motivated by the fact that we are not aware of a single method for rewrite systems which translates a complexity problem into computationally simpler sub-problems. Any proof is of the form

$$\frac{\vdash \mathcal{P}_1 \colon f_1 \quad \cdots \quad \vdash \mathcal{P}_n \colon f_n}{\vdash \mathcal{P} \colon f} \ ,$$

with $f \in \mathcal{O}(f_i)$ for some $i \in \{1, \ldots, n\}$. This implies that the maximal bound one can prove is essentially determined by the strength of the employed base techniques, viz, complexity pairs. In our experience however, a complexity prover is seldom able to synthesise a suitable and precise complexity pair that induces a complexity bound beyond a cubic polynomial.

Decomposition techniques are of utmost importance, and often considered crucial for the (static) resource analysis. For instance, one of the main strengths of the type-based system of Hoffmann et al. [21] is precisely its inherent composability. One strong points in the *size-change abstraction* [39] based analysis of Zuleger et al. [20] is that global runtime bounds of imperative programs can be obtained by combining bounds on *strongly connected components* (*SCCs* for short). The technique proposed here is similar, although not identical, to such a decomposition. We also allow for a separate analysis of SCCs. Taking the call structure into account, a global bounding function is obtained by multiplying bounds on sub-problems. Hence there is also a clear relationship to the *multiplicative composition of ranking functions* introduced by Gulwani and Zuleger [19].

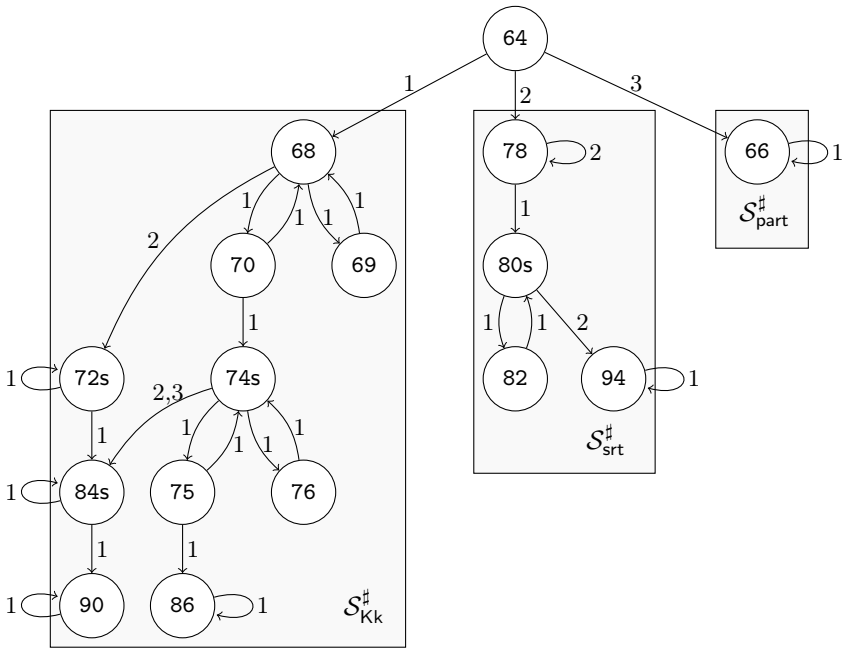


Figure 6: Dependency Graph $\mathcal{G}_{\mathsf{Ks}}$ of DP problem $\langle \mathcal{S}^\sharp_{\mathsf{Ks}}/\mathcal{U}_{\mathsf{K}}, \mathcal{R}_{\mathsf{K}}, \mathcal{T}^\sharp_{\mathsf{K}} \rangle$.

Before we continue, we want to remark that relative decomposition, as given in Theorem 3, can be used to decompose the input problem guided by the dependency graph.

Example 21 (Continued from Example 20). Consider the DG $\mathcal{G}_{\mathsf{Ks}}$ of the DP problem $\langle \mathcal{S}^\sharp_{\mathsf{Ks}}/\mathcal{U}_{\mathsf{K}}, \mathcal{R}_{\mathsf{K}}, \mathcal{T}^\sharp_{\mathsf{K}} \rangle$, depicted in Figure 6. As demarcated by the rectangles in the figure, this graph has three independent subgraphs. Let $\mathcal{S}^\sharp_{\mathsf{Kk}} \subseteq \mathcal{S}^\sharp_{\mathsf{Ks}}$, $\mathcal{S}^\sharp_{\mathsf{srt}} \subseteq \mathcal{S}^\sharp_{\mathsf{Ks}}$ and $\mathcal{S}^\sharp_{\mathsf{part}} \subseteq \mathcal{S}^\sharp_{\mathsf{Ks}}$ denote the set of DPs as indicated in the three components of the dependency graph. Essentially as an application of the relative decomposition processor, the DPs

$\mathcal{S}^\sharp_{\mathsf{Kk}}$, $\mathcal{S}^\sharp_{\mathsf{srt}}$, and $\mathcal{S}^\sharp_{\mathsf{part}}$ can be treated completely independently. To this extend, consider the dependency pairs

$$\begin{aligned}
&\mathtt{64s}\colon \mathsf{forest}^\sharp(\mathsf{graph}(N,E)) \to \mathsf{sort}^\sharp(E)\\
&\mathtt{64k}\colon \mathsf{forest}^\sharp(\mathsf{graph}(N,E)) \to \mathsf{kruskal}^\sharp(\mathsf{sort}(E),[\,],\mathsf{partitions}(N))\\
&\mathtt{64p}\colon \mathsf{forest}^\sharp(\mathsf{graph}(N,E)) \to \mathsf{partitions}^\sharp(N)\;,
\end{aligned}$$

which reflect the three individual calls from $\mathsf{forest}^\sharp$ (via the DP $\mathtt{64}$) to the three components $\mathcal{S}^\sharp_{\mathsf{Kk}}$, $\mathcal{S}^\sharp_{\mathsf{srt}}$, and $\mathcal{S}^\sharp_{\mathsf{part}}$ respectively. Consider the following three sub-problems which are obtained by considering only nodes from the three indicated sub-graphs, together with the individual calls from $\mathsf{forest}^\sharp$:

$$\mathcal{P}^\sharp_{\mathsf{Kk}} := \langle \mathcal{S}^\sharp_{\mathsf{Kk}}/\{\mathtt{64k}\}\cup\mathcal{U}_{\mathsf{K}},\mathcal{R}_{\mathsf{K}},\mathcal{T}^\sharp_{\mathsf{K}}\rangle \quad \mathcal{P}^\sharp_{\mathsf{srt}} := \langle \mathcal{S}^\sharp_{\mathsf{srt}}/\{\mathtt{64s}\}\cup\mathcal{U}_{\mathsf{srt}},\mathcal{R}_{\mathsf{K}},\mathcal{T}^\sharp_{\mathsf{K}}\rangle \quad \mathcal{P}^\sharp_{\mathsf{part}} := \langle \mathcal{S}^\sharp_{\mathsf{part}}/\{\mathtt{64p}\},\mathcal{R}_{\mathsf{K}},\mathcal{T}^\sharp_{\mathsf{K}}\rangle\;.$$

Here $\mathcal{U}_{\mathsf{srt}} := \{\mathtt{7},\mathtt{21}\text{–}\mathtt{26},\mathtt{35}\text{–}\mathtt{38}\} \subseteq \mathcal{U}_{\mathsf{K}}$. We reason below that

$$\frac{\vdash \mathcal{P}^\sharp_{\mathsf{Kk}}\colon n^2 \quad \vdash \mathcal{P}^\sharp_{\mathsf{srt}}\colon n^2 \quad \vdash \mathcal{P}^\sharp_{\mathsf{part}}\colon n^2}{\vdash \langle \mathcal{S}^\sharp_{\mathsf{Ks}}/\mathcal{U}_{\mathsf{K}},\mathcal{R}_{\mathsf{K}},\mathcal{T}^\sharp_{\mathsf{K}}\rangle\colon n^2}$$

is a sound inference. To see this, consider first the following intermediate DP problems

$$\begin{aligned}
\mathcal{P}^\sharp_{\mathsf{frst\text{-}rel}} &:= \langle \{\mathtt{64}\}/\mathcal{S}^\sharp_{\mathsf{Kk}}\cup\mathcal{S}^\sharp_{\mathsf{srt}}\cup\mathcal{S}^\sharp_{\mathsf{part}}\cup\mathcal{U}_{\mathsf{K}},\mathcal{R}_{\mathsf{K}},\mathcal{T}^\sharp_{\mathsf{K}}\rangle \qquad \mathcal{P}^\sharp_{\mathsf{srt\text{-}rel}} := \langle \mathcal{S}^\sharp_{\mathsf{srt}}/\mathcal{S}^\sharp_{\mathsf{Kk}}\cup\mathcal{S}^\sharp_{\mathsf{part}}\cup\{\mathtt{64}\}\cup\mathcal{U}_{\mathsf{K}},\mathcal{R}_{\mathsf{K}},\mathcal{T}^\sharp_{\mathsf{K}}\rangle\\
\mathcal{P}^\sharp_{\mathsf{KK\text{-}rel}} &:= \langle \mathcal{S}^\sharp_{\mathsf{Kk}}/\mathcal{S}^\sharp_{\mathsf{srt}}\cup\mathcal{S}^\sharp_{\mathsf{part}}\cup\{\mathtt{64}\}\cup\mathcal{U}_{\mathsf{K}},\mathcal{R}_{\mathsf{K}},\mathcal{T}^\sharp_{\mathsf{K}}\rangle \qquad \mathcal{P}^\sharp_{\mathsf{part\text{-}rel}} := \langle \mathcal{S}^\sharp_{\mathsf{part}}/\mathcal{S}^\sharp_{\mathsf{Kk}}\cup\mathcal{S}^\sharp_{\mathsf{srt}}\cup\{\mathtt{64}\}\cup\mathcal{U}_{\mathsf{K}},\mathcal{R}_{\mathsf{K}},\mathcal{T}^\sharp_{\mathsf{K}}\rangle\;.
\end{aligned}$$

These problems are obtained from the input problem $\langle \mathcal{S}^\sharp_{\mathsf{Ks}}/\mathcal{U}_{\mathsf{K}},\mathcal{R}_{\mathsf{K}},\mathcal{T}^\sharp_{\mathsf{K}}\rangle$ by moving dependency pairs from the strict to the weak component, in accordance to the partitioning of the DG $\mathcal{G}_{\mathsf{Ks}}$. Note that dependency graphs of these problems coincide with the DG from $\mathcal{G}_{\mathsf{Ks}}$. Three applications of relative decomposition allow us to deduce

$$\frac{\vdash \mathcal{P}^\sharp_{\mathsf{KK\text{-}rel}}\colon n^2 \qquad \dfrac{\vdash \mathcal{P}^\sharp_{\mathsf{srt\text{-}rel}}\colon n^2 \qquad \dfrac{\vdash \mathcal{P}^\sharp_{\mathsf{part\text{-}rel}}\colon n^2 \quad \vdash \mathcal{P}^\sharp_{\mathsf{frst\text{-}rel}}\colon n^2}{\vdash \langle \mathcal{S}^\sharp_{\mathsf{part}}\cup\{\mathtt{64}\}/\mathcal{S}^\sharp_{\mathsf{srt}}\cup\mathcal{S}^\sharp_{\mathsf{Kk}}\cup\mathcal{U}_{\mathsf{K}},\mathcal{R}_{\mathsf{K}},\mathcal{T}^\sharp_{\mathsf{K}}\rangle\colon n^2}}{\vdash \langle \mathcal{S}^\sharp_{\mathsf{srt}}\cup\mathcal{S}^\sharp_{\mathsf{part}}\cup\{\mathtt{64}\}/\mathcal{S}^\sharp_{\mathsf{Kk}}\cup\mathcal{U}_{\mathsf{K}},\mathcal{R}_{\mathsf{K}},\mathcal{T}^\sharp_{\mathsf{K}}\rangle\colon n^2}}{\vdash \langle \mathcal{S}^\sharp_{\mathsf{Ks}}/\mathcal{U}_{\mathsf{K}},\mathcal{R}_{\mathsf{K}},\mathcal{T}^\sharp_{\mathsf{K}}\rangle\colon n^2}$$

Consider the intermediate problem $\mathcal{P}^\sharp_{\mathsf{frst\text{-}rel}}$. Note that since the DP $\mathtt{64}$ has no incoming edges in the DG, i.e. $\mathsf{Pre}_{\mathcal{G}_{\mathsf{Ks}}}(\{\mathtt{64}\}) = \varnothing$, the complexity function of $\mathcal{P}^\sharp_{\mathsf{frst\text{-}rel}}$ is even constant. We can show this by one application of the predecessor estimation processor and the axiom empty.

$$\dfrac{\dfrac{}{\vdash \langle \varnothing/\mathcal{S}^\sharp_{\mathsf{Kk}}\cup\mathcal{S}^\sharp_{\mathsf{srt}}\cup\mathcal{S}^\sharp_{\mathsf{part}}\cup\{\mathtt{64}\}\cup\mathcal{U}_{\mathsf{K}},\mathcal{R}_{\mathsf{K}},\mathcal{T}^\sharp_{\mathsf{K}}\rangle\colon n^2}\;\mathsf{empty}}{\vdash \mathcal{P}^\sharp_{\mathsf{frst\text{-}rel}}\colon n^2}\;\mathsf{Predecessor\ Estimation}$$

Consider now the intermediate DP problem $\mathcal{P}^\sharp_{\mathsf{srt\text{-}rel}}$. The DPs $\mathcal{S}^\sharp_{\mathsf{Kk}}$ and $\mathcal{S}^\sharp_{\mathsf{part}}$ constitute a forward closed set of weak DPs in $\mathcal{P}^\sharp_{\mathsf{srt\text{-}rel}}$. So they can be simply dropped from $\mathcal{P}^\sharp_{\mathsf{srt\text{-}rel}}$, by Theorem 8. As a consequence, the right-hand side of the DP $\mathtt{64}$ can be simplified to

$$\mathtt{64s}\colon \mathsf{forest}^\sharp(\mathsf{graph}(N,E)) \to \mathsf{sort}^\sharp(E)\;,$$

by Theorem 10. Finally, the set of rewrite rules can be narrowed to the usable rules $\mathcal{U}_{\mathsf{srt}}$, by Theorem 7. In total, this proves $\mathcal{P}^\sharp_{\mathsf{srt}}\colon n^2 \vdash \mathcal{P}^\sharp_{\mathsf{srt\text{-}rel}}\colon n^2$:

$$\dfrac{\dfrac{\dfrac{\vdash \langle \mathcal{S}^\sharp_{\mathsf{srt}}/\{\mathsf{64s}\} \cup \mathcal{U}_{\mathsf{srt}}, \mathcal{R}_\mathsf{K}, \mathcal{T}^\sharp_\mathsf{K}\rangle\colon n^2}{\vdash \langle \mathcal{S}^\sharp_{\mathsf{srt}}/\{\mathsf{64s}\} \cup \mathcal{U}_\mathsf{K}, \mathcal{R}_\mathsf{K}, \mathcal{T}^\sharp_\mathsf{K}\rangle\colon n^2}\ \text{Usable Rules}}{\vdash \langle \mathcal{S}^\sharp_{\mathsf{srt}}/\{\mathsf{64}\} \cup \mathcal{U}_\mathsf{K}, \mathcal{R}_\mathsf{K}, \mathcal{T}^\sharp_\mathsf{K}\rangle\colon n^2}\ \text{Simpl. DP-RHS}}{\vdash \langle \mathcal{S}^\sharp_{\mathsf{srt}}/\mathcal{S}^\sharp_{\mathsf{Kk}} \cup \mathcal{S}^\sharp_{\mathsf{part}} \cup \{\mathsf{64}\} \cup \mathcal{U}_\mathsf{K}, \mathcal{R}_\mathsf{K}, \mathcal{T}^\sharp_\mathsf{K}\rangle\colon n^2}\ \text{Remove Weak Suffixes}$$

By identical reasoning, we can prove $\mathcal{P}^\sharp_{\mathsf{Kk}}\colon n^2 \vdash \mathcal{P}^\sharp_{\mathsf{KK\text{-}rel}}\colon n^2$, and likewise $\mathcal{P}^\sharp_{\mathsf{part}}\colon n^2 \vdash \mathcal{P}^\sharp_{\mathsf{part\text{-}rel}}\colon n^2$. Putting these proofs together yields the inference outlined in the beginning of the example.

This form of decomposition however fails to achieve one main motivation. The complexity of the input problem is reflected in the complexity of at least one of the sub-problems. In the example above, the complexity of $\mathcal{P}^\sharp_{\mathsf{Kk}}$ and $\mathcal{P}^\sharp_{\mathsf{srt}}$ is bounded by a quadratic polynomial from below.

In contrast, dependency graph decomposition seeks to analyse *recursive definitions*, reflected by *cycles* in the DG, separately. Consider for instance the DP problem $\mathcal{P}^\sharp_{\mathsf{srt}} = \langle \mathcal{S}^\sharp_{\mathsf{srt}}/\{\mathsf{64s}\} \cup \mathcal{U}_{\mathsf{srt}}, \mathcal{R}_\mathsf{K}, \mathcal{T}^\sharp_\mathsf{K}\rangle$ obtained in Example 21, involving the dependency concerning the sorting algorithm underlying $\mathcal{R}_\mathsf{K}$. Let $[e_1, \cdots, e_n]$ abbreviate a list of edges $e_1 :: \cdots :: (e_n :: [\,])$. To estimate the complexity of $\mathcal{P}^\sharp_{\mathsf{srt}}$, without loss of generality we fix a derivation starting from $t^\sharp := \mathsf{sort}^\sharp([e_1, \cdots, e_n])$. The part of the DG of $\mathcal{P}^\sharp_{\mathsf{srt}}$ relevant for the reduction of $t^\sharp$ consists of the DPs 78, defining $\mathsf{sort}^\sharp$, and $\mathcal{S}^\sharp_{\mathsf{insert}} := \{\mathsf{80s}, \mathsf{82}, \mathsf{94}\}$. Edges are as depicted in Figure 6. A reduction of $t^\sharp$ involves one recursion with respect to the DP 78. As indicated by the edge from 78 to 80s, each recursion (possibly) triggers a sub-derivation that involves the DP 80s. Indeed, a derivation tree of $t^\sharp$ can be sketched as in Figure 7. The terms $t^\sharp_i$ demarcate the first application of the DP 80s, i.e. the call to $\mathsf{insert}^\sharp$. As a consequence of Lemma 9 and the shape of the DG, the gray subtrees T_i ($i = 1, \ldots, n$) contain only application of DPs $\mathcal{S}^\sharp_{\mathsf{insert}}$. More precise, the trees T_i are derivation trees with respect to the problem $\mathcal{P}^\sharp_l := \langle \mathcal{S}^\sharp_{\mathsf{insert}}/\mathcal{U}_{\mathsf{srt}}, \mathcal{R}_\mathsf{K}, \mathcal{T}^\sharp_\mathsf{K}\rangle$. Dual, the white part T_u reflects the recursion involving the DP 78, that is, T_u is a derivation tree of with respect to the problem $\mathcal{P}^\sharp_u := \langle \{\mathsf{78}\}/\mathcal{U}_{\mathsf{srt}}, \mathcal{R}_\mathsf{K}, \mathcal{T}^\sharp_\mathsf{K}\rangle$. In principle, it should thus be possible to divide $\mathcal{P}^\sharp_{\mathsf{srt}}$ into $\mathcal{P}^\sharp_l$ and $\mathcal{P}^\sharp_u$, and propagate bounds on the complexity of $\mathcal{P}^\sharp_l$ and $\mathcal{P}^\sharp_l$ back to $\mathcal{P}^\sharp_{\mathsf{srt}}$.

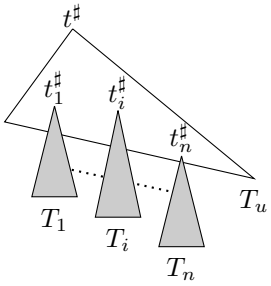


Figure 7: Separation of derivation tree T in upper and lower layer.

Observe that both problems admit linear complexity. This can be easily verified by tool TcT. Note that in our example, the term $t^\sharp_i$ ($i = 1, \ldots, n$) are of the form $\mathsf{insert}^\sharp(e_i, [e'_{i+1}, \ldots, e'_n]) \in \mathcal{T}^\sharp_\mathsf{K}$, where $[e'_{i+1}, \ldots, e'_n]$ is the (unique) normal form of $\mathsf{sort}([e_{i+1}, \ldots, e_n])$. Thus the application of DPs in a subtree T_i is bounded by a linear function, in the size of $t^\sharp_i$ and thus $t^\sharp = \mathsf{sort}^\sharp([e_1, \ldots, e_n])$. The number of applications of DPs in T_u is bounded by a linear function in the size of $t^\sharp$. As this gives also a bound on the number of trees T_i, we obtain overall a quadratic bound on the number of applications of DPs in the full tree T.

The dependency graph decomposition processor formalises this kind of reasoning. Fix a DP problem $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp \cup \mathcal{S}/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$. The separation of derivation trees T of $\mathcal{P}^\sharp$ into *upper components* T_u and *lower components* (consisting of trees $T_1, \ldots, T_n$) is given by a partitioning of strict dependency pairs $\mathcal{S}^\sharp = \mathcal{S}^\sharp_u \cup \mathcal{S}^\sharp_l$ and weak dependency pairs $\mathcal{W}^\sharp = \mathcal{W}^\sharp_u \cup \mathcal{W}^\sharp_l$, such that $\mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l$ is forward closed in the DG of $\mathcal{P}^\sharp$. For instance, the set $\mathcal{S}^\sharp_{\mathsf{insert}}$ used above forms a forward closed in $\mathcal{P}^\sharp_{\mathsf{srt}}$. For a derivation tree T of $\mathcal{P}^\sharp$, the separation of DPs induces a separation of T into two (possibly empty) layers, demarcated by topmost applications of DPs from $\mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l$ as indicated in Figure 7. The *lower layer* consists of the (maximal) subtrees $T_1, \ldots, T_n$ of T with a dependency pair $l \to r \in \mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l$ applied at the root. The *upper layer* consists of the derivation tree T_u obtained from T by removing the sub-trees $T_1, \ldots, T_n$. Figure 7 illustrates this separation.

Define $\mathcal{P}^\sharp_u := \langle \mathcal{S}^\sharp_u \cup \mathcal{S}/\mathcal{W}^\sharp_u \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$ and $\mathcal{P}^\sharp_l := \langle \mathcal{S}^\sharp_l \cup \mathcal{S}/\mathcal{W}^\sharp_l \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$. By construction, T_u is a $\mathcal{P}^\sharp_u$ derivation tree of $t^\sharp$, i.e. contains no application of DPs from $\mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l$. The complexity function of

$\mathcal{P}_u^\sharp$ thus binds the number of applications of DPs and rules from $\mathcal{S}^\sharp \cup \mathcal{S}$ in T_u, in the size of $t^\sharp$. Dual, by forward closure of $\mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp$ the $\mathcal{P}^\sharp$ derivations trees T_i from the lower layer are $\mathcal{P}_l^\sharp$ derivation trees of terms $t_i^\sharp$. However, the complexity function of $\mathcal{P}_l^\sharp$ may not suitably estimate the size of the derivation trees of $t_i^\sharp$, more precise $|T_i|_{\mathcal{S}_l^\sharp \cup \mathcal{S}}$, in the size of $t^\sharp$. On the one hand, $t_i^\sharp \in \mathcal{T}^\sharp$ does not hold in general. On the other hand, the size of $t_i^\sharp$ does not necessarily relate linearly (even not polynomially) to the size of $t^\sharp$. To resolve these issues, DG decomposition adds suitable dependency pairs $\mathsf{sep}(\mathcal{S}_u^\sharp \cup \mathcal{W}_u^\sharp)$ to $\mathcal{P}_l^\sharp$, defined as follows.

Definition 18. For a set of DPs $\mathcal{R}^\sharp$ we define

$$\mathsf{sep}(\mathcal{R}^\sharp) := \{l \to r_i \mid l \to \mathrm{COM}(r_1, \ldots, r_i, \ldots, r_k) \in \mathcal{R}^\sharp\} \; .$$

These DPs are used to extend the derivation tree T_i of $t_i^\sharp$ to a derivation tree of $t^\sharp \in \mathcal{T}^\sharp$.

Example 22 (Continued from Example 21). Consider the DP problem $\mathcal{P}_{\mathsf{srt}}^\sharp = \langle \mathcal{S}_{\mathsf{srt}}^\sharp / \{\mathtt{64s}\} \cup \mathcal{U}_{\mathsf{srt}}, \mathcal{R}_{\mathsf{K}}, \mathcal{T}_{\mathsf{K}}^\sharp \rangle$ together with the forward closed set of DPs $\mathcal{S}_{\mathsf{insert}}^\sharp := \{\mathtt{80s}, \mathtt{82}, \mathtt{94}\}$. Let T denote a $\mathcal{P}_{\mathsf{srt}}^\sharp$ derivation tree of $\mathsf{sort}^\sharp([e_1, \ldots, e_n])$ for edges $e_1, \ldots, e_n$. The nodes labeled by $\mathsf{insert}^\sharp(e_i, [e'_{i+1}, \ldots, e'_n])$, where $[e'_{i+1}, \ldots, e'_n]$ the sorted list of $[e_{i+1}, \ldots, e_n]$, demarcate the upper T_u and lower layer in T $(i = 1, \ldots, n)$. Then $\mathsf{sep}(\{\mathtt{64s}, \mathtt{78}\})$ consists of $\mathtt{64s}$ together with the DPs

$$\mathtt{78a}\colon \mathsf{sort}^\sharp(e :: E) \to \mathsf{insert}^\sharp(e, \mathsf{sort}(E)) \qquad \mathtt{78b}\colon \mathsf{sort}^\sharp(e :: E) \to \mathsf{sort}^\sharp(E) \; .$$

Observe that in combination with $\mathcal{U}_{\mathsf{srt}}$, these can be used to generate the terms $\mathsf{insert}^\sharp(e_i, [e'_{i+1}, \ldots, e'_n])$ $(i = 1, \ldots, n)$ from the initial term $\mathsf{sort}^\sharp([e_1, \ldots, e_n])$. As a consequence, the complexity problem

$$\mathcal{P}_{\mathsf{insert}}^\sharp := \langle \mathcal{S}_{\mathsf{insert}}^\sharp / \{\mathtt{64s}, \mathtt{78a}, \mathtt{78b}\} \cup \mathcal{U}_{\mathsf{srt}}, \mathcal{R}_{\mathsf{K}}, \mathcal{T}_{\mathsf{K}}^\sharp \rangle \; ,$$

accounts for applications of DPs from $\mathcal{S}_{\mathsf{insert}}^\sharp$ in a sub-tree T_i $(i = 1, \ldots, n)$. Conversely, the DP problem $\mathcal{P}_{\mathsf{srt}}^\sharp = \langle \{\mathtt{78}\} / \{\mathtt{64s}\} \cup \mathcal{U}_{\mathsf{srt}}, \mathcal{R}_{\mathsf{K}}, \mathcal{T}_{\mathsf{K}}^\sharp \rangle$ accounts for the application of the DPs in the upper component T_u.

The next two technical lemmas formalise the central proof steps carried out in the informal reasoning from above.

Lemma 11. *Consider a DP problem $\mathcal{P}^\sharp = \langle \mathcal{S}_u^\sharp \cup \mathcal{S}_l^\sharp \cup \mathcal{S} / \mathcal{W}_u^\sharp \cup \mathcal{W}_l^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$. Let $\mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp$ be a forward closed set of DPs in $\mathcal{P}^\sharp$, and let T be a $\mathcal{P}^\sharp$ derivation tree T of $t^\sharp \in \mathcal{T}^\sharp$. Consider the maximal sub-trees $T_1, \ldots, T_n$ of T such that a DP from $\mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp$ is applied at the root, and let T_u be obtained from T by removing $T_1, \ldots, T_n$. Then*

1. *T_u is a $\langle \mathcal{S}_u^\sharp \cup \mathcal{S} / \mathcal{W}_u^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$ derivation tree of $t^\sharp$;*
2. *for each $i = 1, \ldots, n$, there exists a $\langle \mathcal{S}_l^\sharp \cup \mathcal{S} / \mathcal{W}_l^\sharp \cup \mathcal{W} \cup \mathsf{sep}(\mathcal{S}_u^\sharp \cup \mathcal{W}_u^\sharp), \mathcal{Q}, \mathcal{T}^\sharp \rangle$ derivation tree of $t^\sharp$, that contains T_i as sub-tree.*

Proof. The first assertion follows by definition. We consider the second assertion. Observe that on the path from the root of T to T_i, by construction only dependency pairs $l^\sharp \to \mathrm{COM}(r_1^\sharp, \ldots, r_k^\sharp) \in \mathcal{S}_u^\sharp \cup \mathcal{W}_u^\sharp$ are applied. Replacing each application of $l^\sharp \to \mathrm{COM}(r_1^\sharp, \ldots, r_k^\sharp)$ by a corresponding DP $l^\sharp \to r_i^\sharp \in \mathsf{sep}(\mathcal{S}_u^\sharp \cup \mathcal{W}_u^\sharp)$ yields a $\langle \mathcal{S}_u^\sharp \cup \mathcal{S}_l^\sharp \cup \mathcal{S} / \mathcal{W}_u^\sharp \cup \mathcal{W}_l^\sharp \cup \mathcal{W} \cup \mathsf{sep}(\mathcal{S}_u^\sharp \cup \mathcal{W}_u^\sharp), \mathcal{Q}, \mathcal{T}^\sharp \rangle$ derivation tree of $t^\sharp$. This tree contains T_i as sub-tree. Since $\mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp$ is forward closed, Lemma 9 yields that T_i contains only applications of DPs from $\mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp$, besides application of rules $l \to r \in \mathcal{S} \cup \mathcal{W}$. Hence the constructed tree is even a $\langle \mathcal{S}_l^\sharp \cup \mathcal{S} / \mathcal{W}_l^\sharp \cup \mathcal{W} \cup \mathsf{sep}(\mathcal{S}_u^\sharp \cup \mathcal{W}_u^\sharp), \mathcal{Q}, \mathcal{T}^\sharp \rangle$ derivation tree. □

Lemma 12. *Consider a DP problem $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp \cup \mathcal{S} / \mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$. Let $\mathcal{R}^\sharp$ be a forward closed set of DPs in $\mathcal{P}^\sharp$, and let T be a $\mathcal{P}^\sharp$ derivation tree T of $t^\sharp \in \mathcal{T}^\sharp$. Let $T_1, \ldots, T_n$ denote the maximal sub-trees of T with $l \to r \in \mathcal{R}^\sharp$ applied at the root. There exists a constant $\Delta \in \mathbb{N}$ depending only on $\mathcal{P}^\sharp$ such that $n \leqslant \max\{1, |T|_{\mathsf{Pre}_{\mathcal{G}}(\mathcal{R}^\sharp) \setminus \mathcal{R}^\sharp} \cdot \Delta\}$.*

Proof. The proof is a slight variation of the proof of Lemma 10. Let Δ be the maximal arity of a compound symbol from $\mathcal{P}^\sharp$, and observe that every node in T has at most Δ successors. Denote by $\{u_1, \dots, u_n\}$ the roots of T_i $(i = 1, \dots, n)$. The non-trivial case is $n > 1$. In this case, each path from the root of T to the nodes $u_i \in \{u_1, \dots, u_n\}$ contains at least one node with a DP applied. Let $\{v_1, \dots, v_m\}$ collect such nodes closest to $\{u_1, \dots, u_n\}$. In particular, we can thus associate to every node $u_i \in \{u_1, \dots, u_n\}$ a node $v_{i'} \in \{v_1, \dots, v_m\}$ and DP $l \to r \in \mathcal{P}^\sharp$ such that $v_{i'} \xrightarrow{\{l_i \to r_i\}}_T \cdot \xrightarrow{\mathcal{S} \cup \mathcal{W}}{}^*_T u_i$ holds. As $v_{i'}$ has at most Δ successors and $\xrightarrow{\mathcal{S} \cup \mathcal{W}}_T$ is non-branching, it follows that $n \leqslant \Delta \cdot m$. By Lemma 9, for $i = 1, \dots, n$ we see $l_i \to r_i \in \mathsf{Pre}_\mathcal{G}(\mathcal{R}^\sharp)$. As T_i is maximal, $l_i \to r_i \notin \mathcal{R}^\sharp$. Hence $m \leqslant |T|_{\mathsf{Pre}_\mathcal{G}(\mathcal{R}^\sharp) \setminus \mathcal{R}^\sharp}$ and the lemma follows. □

Theorem 11 (Dependency Graph Decomposition Processor)**.** *Consider a DP problem $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp_u \cup \mathcal{S}^\sharp_l \cup \mathcal{S}/\mathcal{W}^\sharp_u \cup \mathcal{W}^\sharp_l \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$ together with an approximated dependency graph $\mathcal{G}$ of $\mathcal{P}^\sharp$ such that:*

1. *$\mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l$ is forward closed in $\mathcal{G}$, and*
2. *$\mathsf{Pre}_\mathcal{G}(\mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l) \cap \mathcal{W}^\sharp_u = \varnothing$.*

The following processor is sound.

$$\frac{\vdash \langle \mathcal{S}^\sharp_u \cup \mathcal{S}/\mathcal{W}^\sharp_u \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle \colon f \quad \vdash \langle \mathcal{S}^\sharp_l \cup \mathcal{S}/\mathcal{W}^\sharp_l \cup \mathsf{sep}(\mathcal{S}^\sharp_u \cup \mathcal{W}^\sharp_u) \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle \colon g}{\vdash \langle \mathcal{S}^\sharp_u \cup \mathcal{S}^\sharp_l \cup \mathcal{S}/\mathcal{W}^\sharp_u \cup \mathcal{W}^\sharp_l \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle \colon f * g} \ ,$$

for all functions f and g such that $f(n) \neq 0$ and $g(n) \neq 0$. Here $f * g$ denotes the function h defined by $h(n) := f(n) \cdot g(n)$.

Proof. Consider a $\mathcal{P}^\sharp$ derivation tree of $t^\sharp \in \mathcal{T}^\sharp$. We estimate $|T|_{\mathcal{S}^\sharp_u \cup \mathcal{S}^\sharp_l \cup \mathcal{S}}$ by a function in $\mathcal{O}(f * g)$, tacitly employing Lemma 8. Consider the separation of T as induced by forward closure of $\mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l$ into the upper layer T_u, and lower layer that consists of the derivation trees T_i of $t^\sharp_i$ $(i = 1, \dots, n)$. By Lemma 11(2) the trees T_i $(i = 1, \dots, m)$ can be extended to $\langle \mathcal{S}^\sharp_l \cup \mathcal{S}/\mathcal{W}^\sharp_l \cup \mathcal{W} \cup \mathsf{sep}(\mathcal{S}^\sharp_u \cup \mathcal{W}^\sharp_u), \mathcal{Q}, \mathcal{T}^\sharp \rangle$ derivation tree T'_i of $t^\sharp$. In particular, the complexity of $\langle \mathcal{S}^\sharp_l \cup \mathcal{S}/\mathcal{W}^\sharp_l \cup \mathcal{W} \cup \mathsf{sep}(\mathcal{S}^\sharp_u \cup \mathcal{W}^\sharp_u), \mathcal{Q}, \mathcal{T}^\sharp \rangle$ binds applications of $\mathcal{S}^\sharp_l \cup \mathcal{S}$ in T_i, i.e. $|T_i|_{\mathcal{S}^\sharp_l \cup \mathcal{S}} = |T'_i|_{\mathcal{S}^\sharp_l \cup \mathcal{S}}$. Hence $|T_i|_{\mathcal{S}^\sharp_l \cup \mathcal{S}} \in \mathcal{O}(g(|t^\sharp|))$ by the second pre-condition of the processor. Similar, Lemma 11(1) and the first precondition of the processor gives $|T_u|_{\mathcal{S}^\sharp_u \cup \mathcal{S}} \in \mathcal{O}(f(|t^\sharp|))$. By assumption 2 and Lemma 12 we see $n \leqslant \max\{1, |T|_{\mathsf{Pre}_\mathcal{G}(\mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l) \setminus (\mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l)}\} \leqslant \max\{1, |T_u|_{\mathcal{S}^\sharp_u \cup \mathcal{S}}\}$. Putting these bounds together we get

$$\begin{aligned}
|T|_{\mathcal{S}^\sharp_u \cup \mathcal{S}^\sharp_l \cup \mathcal{S}} &= |T_u|_{\mathcal{S}^\sharp \cup \mathcal{S}} + \textstyle\sum_{i=1}^n |T_i|_{\mathcal{S}^\sharp \cup \mathcal{S}} \\
&= |T_u|_{\mathcal{S}^\sharp_u \cup \mathcal{S}} + \textstyle\sum_{i=1}^n |T_i|_{\mathcal{S}^\sharp_l \cup \mathcal{S}} \\
&\leqslant |T_u|_{\mathcal{S}^\sharp_u \cup \mathcal{S}} + \max\{1, |T_u|_{\mathcal{S}^\sharp_u \cup \mathcal{S}}\} \cdot \max_{i=1}^n |T_i|_{\mathcal{S}^\sharp_l \cup \mathcal{S}} \\
&\in \mathcal{O}(f(|t^\sharp|)) + \mathcal{O}(f(|t^\sharp|)) \cdot \mathcal{O}(g(|t^\sharp|)) = \mathcal{O}(h(|t^\sharp|)) \ .
\end{aligned}$$

□

The next example shows that the DP problem $\mathcal{P}^\sharp_{\mathsf{srt}}$ has quadratic complexity, using Theorem 11.

Example 23 (Continued from Example 22). Consider the DP problem $\mathcal{P}^\sharp_{\mathsf{srt}}$ from Example 21, and let $\mathcal{G}$ denote the DG of $\mathcal{P}^\sharp_{\mathsf{srt}}$. We already observed that the set DPs $\mathcal{S}^\sharp_{\mathsf{insert}} = \{\mathsf{80s}, \mathsf{82}, \mathsf{94}\}$ is forward closed in $\mathcal{P}^\sharp_{\mathsf{srt}}$. As no DP in $\mathsf{Pre}_\mathcal{G}(\mathcal{S}^\sharp_{\mathsf{insert}}) = \{\mathsf{78}\}$ occurs in the weak component of $\mathcal{P}^\sharp_{\mathsf{srt}}$, also the second precondition of Theorem 11 is satisfied, and thus

$$\frac{\vdash \langle \{\mathsf{78}\}/\{\mathsf{64s}\} \cup \mathcal{U}_{\mathsf{srt}}\}, \mathcal{R}_\mathsf{K}, \mathcal{T}^\sharp_\mathsf{K} \rangle \colon n \quad \vdash \mathcal{P}^\sharp_{\mathsf{insert}} \colon n}{\vdash \mathcal{P}^\sharp_{\mathsf{srt}} \colon n^2} \ .$$

Using the simplifications presented in Theorem 10 and Theorem 7, together with suitable complexity pairs (Theorem 2), it is not difficult to prove the assumptions of this inference. We conclude $\vdash \mathcal{P}^\sharp_{\mathsf{srt}} \colon n^2$ by Theorem 11.

We want to remark that DG decomposition can overestimate the complexity, i.e. DG decomposition is not complete. For this reason it is not always beneficial to exhaustively apply this processor.

Example 24 (Continued from Example 23). The set $\{94\}$ constitutes a forward closed set of DPs in the DP problem $\mathcal{P}^\sharp_{\mathsf{insert}}$ depicted in Example 22. Consider the two DPs

$$\texttt{80a}\colon\ \mathsf{insert}^\sharp(e, f :: E) \to \mathsf{insert}?^\sharp(\mathsf{wt}(e) \leqslant \mathsf{wt}(f), e, f, E) \qquad \texttt{80b}\colon\ \mathsf{insert}^\sharp(e, f :: E) \to \mathsf{wt}(e) \leqslant^\sharp \mathsf{wt}(f)\ .$$

Then we have

$$\mathsf{sep}(\{64\mathsf{s}, 78\mathsf{a}, 78\mathsf{b}, 80\mathsf{s}, 82\}) = \{64\mathsf{s}, 78\mathsf{a}, 78\mathsf{b}, 80\mathsf{a}, 80\mathsf{b}, 82\}\ .$$

Applying the DG decomposition processor yields thus sub-problems

$$\langle \{80\mathsf{s}, 82\}/\{64\mathsf{s}, 78\mathsf{a}, 78\mathsf{b}\} \cup \mathcal{U}_{\mathsf{srt}}, \mathcal{R}_{\mathsf{K}}, \mathcal{T}^\sharp_{\mathsf{K}} \rangle\ ,$$

for the upper, and

$$\langle \{94\}/\{64\mathsf{s}, 78\mathsf{a}, 78\mathsf{b}, 80\mathsf{a}, 80\mathsf{b}, 82\} \cup \mathcal{U}_{\mathsf{srt}}, \mathcal{R}_{\mathsf{K}}, \mathcal{T}^\sharp_{\mathsf{K}} \rangle\ ,$$

for the lower component. The complexity of both problems is bounded from below by a linear function. Hence employing DG decomposition, we can only prove a quadratic upper bound on the complexity of $\mathcal{P}^\sharp_{\mathsf{insert}}$, whereas the complexity of $\mathcal{P}^\sharp_{\mathsf{insert}}$ is in fact linear.

Finally, we remark that iterated application of Theorem 11 extends to a separate analysis of all cycles. Hence our method is closely connected to *cycle analysis* as introduced for termination in [35]. Unlike for cycle analysis, dependency graph decomposition takes the call-structure between cycles into account. As demonstrated in the next example, this is essential in the context of complexity analysis.

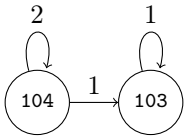
Example 25. Let $\mathcal{P}^\sharp_{\mathsf{exp}} := \langle \mathcal{R}^\sharp_{\mathsf{exp}}/\mathcal{R}_{\mathsf{exp}}, \mathcal{R}_{\mathsf{exp}}, \mathcal{T}^\sharp_{\mathsf{b}} \rangle$ where the dependency pairs $\mathcal{R}^\sharp_{\mathsf{exp}}$ are

$$\texttt{103}\colon\ \mathsf{d}^\sharp(\mathsf{s}(x)) \to \mathsf{d}^\sharp(x) \qquad \texttt{104}\colon\ \mathsf{e}^\sharp(\mathsf{s}(x)) \to \mathsf{c}_2(\mathsf{d}^\sharp(\mathsf{e}(x)), \mathsf{e}^\sharp(x))\ ,$$

and the rewrite system $\mathcal{R}_{\mathsf{exp}}$ is given by the four rules

$$\texttt{105}\colon\ \mathsf{d}(0) \to 0 \qquad \texttt{106}\colon\ \mathsf{d}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{d}(x))) \qquad \texttt{107}\colon\ \mathsf{e}(0) \to \mathsf{s}(0) \qquad \texttt{108}\colon\ \mathsf{e}(\mathsf{s}(x)) \to \mathsf{d}(\mathsf{e}(x))\ ,$$

that compute exponentiation on numerals. The following depicts the DG of $\mathcal{P}^\sharp_{\mathsf{exp}}$.



A decomposition of the dependency pairs in $\mathcal{P}^\sharp_{\mathsf{exp}}$ into cycles, as in termination analysis [35], amounts in our setting to an inference

$$\frac{\vdash \langle \{104\}/\mathcal{R}_{\mathsf{exp}}, \mathcal{R}_{\mathsf{exp}}, \mathcal{T}^\sharp_{\mathsf{b}} \rangle\colon f \quad \vdash \langle \{103\}/\mathcal{R}_{\mathsf{exp}}, \mathcal{R}_{\mathsf{exp}}, \mathcal{T}^\sharp_{\mathsf{b}} \rangle\colon g}{\vdash \langle \{103, 104\}/\mathcal{R}_{\mathsf{exp}}, \mathcal{R}_{\mathsf{exp}}, \mathcal{T}^\sharp_{\mathsf{b}} \rangle\colon h}\ .$$

While the complexity of $\langle \{103\}/\mathcal{R}_{\mathsf{exp}}, \mathcal{R}_{\mathsf{exp}}, \mathcal{T}^\sharp_{\mathsf{b}} \rangle$ and $\langle \{104\}/\mathcal{R}_{\mathsf{exp}}, \mathcal{R}_{\mathsf{exp}}, \mathcal{T}^\sharp_{\mathsf{b}} \rangle$ is linear, the complexity of $\mathcal{P}^\sharp_{\mathsf{exp}}$ is exponential. On the other hand, DG decomposition extends the sub-problem $\langle \{103\}/\mathcal{R}_{\mathsf{exp}}, \mathcal{R}_{\mathsf{exp}}, \mathcal{T}^\sharp_{\mathsf{b}} \rangle$ obtained from the lower cycle with the two DPs

$$\texttt{104a}\colon\ \mathsf{e}^\sharp(\mathsf{s}(x)) \to \mathsf{d}^\sharp(\mathsf{e}(x)) \qquad \texttt{104b}\colon\ \mathsf{e}^\sharp(\mathsf{s}(x)) \to \mathsf{e}^\sharp(x)\ .$$

The so obtained sub-problem $\langle \{103\}/\{104\mathsf{a}, 104\mathsf{b}\} \cup \mathcal{R}_{\mathsf{exp}}, \mathcal{R}_{\mathsf{exp}}, \mathcal{T}^\sharp_{\mathsf{b}} \rangle$ has exponential complexity.

5. Implementation Issues

The proposed framework forms the backbone of our complexity tool TCT, the *Tyrolean Complexity Tool*.[6] TCT is a highly configurable, state-of-the-art complexity analyser for term rewrite system. We kindly refer the reader to the system description [17] for details on usage and features of TCT. As indicated by the results of the *annual international termination competition*,[7] which features four dedicated categories for the complexity analysis of term rewrite systems, TCT is competitive on an international level. In the recent full run of the competition[8] our tool scored highest in three out of the four categories, both in the number of solved systems and precision of the certificate (as indicated by the global score).

Our tool features currently 23 techniques for runtime and, where applicable, for derivational complexity analysis. The proposed framework allows us to seamlessly combine all implemented techniques in our tool. It goes without saying that TCT is not blindly applying the implemented methods to construct a proof tree. The search space would simply be too high. It is clear that complete processors should be preferred over incomplete ones, since any application of an incomplete processor reduces the chance to prove a tight bound on the complexity. There is however theoretically no preferable strategy to combine complete processors. In practice, it makes sense to employ a dependency pair transformation (Theorem 5 or Theorem 6, respectively) first, because these enable the various simplifications discussed in Section 4.4 and Section 4.5. Also, it is preferable to apply syntactic methods before the notoriously slow semantic techniques.

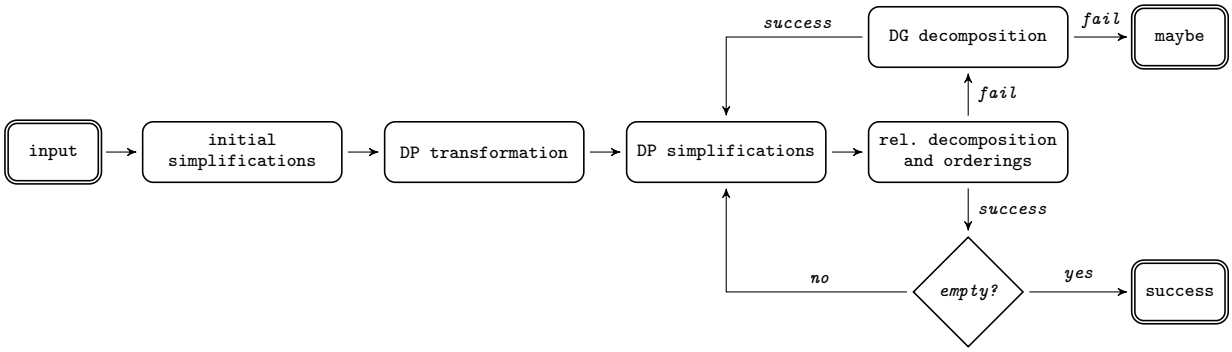


Figure 8: Schematic presentation of the default proof-search strategy DGD underlying TCT as flow chart.

Figure 8 outlines the current default proof-search strategy underlying TCT. Here, boxes correspond to the following sub-strategies. Our experiments confirm that this strategy works effectively in the majority of cases we tested, in terms of the number of proven examples and the precision of the deduced bounds.

- *Initial simplifications:* Initially, the criterion of Hirokawa et al. [40] is used to transform a complexity problem to the corresponding innermost complexity problem, if applicable. Also, on innermost complexity problems some trivial simplifications are employed that possibly remove rules which are inapplicable during reductions.

- *DP transformation:* For runtime complexity analysis, the weak dependency pairs processor (Theorem 5) is employed to transform the current problem into a DP problem. This processor is coupled with the *weightgap condition* [5], under which all rewrite rules from the strict component can be moved to the weak component. For a given innermost runtime complexity problem, the dependency tuples processor (Theorem 6) might also be used. However, the application of dependency tuples is problematic, since the processor is in general not complete. But on the other hand, also the application of the weak

[6] TCT is released under the *GNU Lesser General Public License* (*LGPL*) Version 3. A web-interface and sources can be found online at `http://cl-informatik.uibk.ac.at/software/tct/`.

[7] C.f. `http://termination-portal.org/wiki/Termination_Competition`.

[8] Results available at `http://termcomp.uibk.ac.at/termcomp/competition/competitionSummary.seam?comp=455446`.

dependency pair processor is problematic whenever the weightgap condition cannot be established. The presence of rewrite rules besides DPs in the strict component blocks the application of various techniques employed later. Most notably complexity pairs in the form of safe reduction pairs (compare Lemma 6), but also various DP simplification techniques, are blocked.

In practice, it seems that the latter problem out weights incompleteness of dependency tuples. As compromise, in the current strategy dependency tuples are preferred over weak dependency pairs whenever the weightgap condition cannot be established.

- *DP simplifications:* At this stage the strategy chains various DP simplification techniques. The predecessor estimation and remove weak suffix processors (Theorem 9 and Theorem 8 respectively) are used to delete dependency pairs, compare Example 19. If applicable, the current DP problem is decomposed using the relative decomposition processor (Theorem 3) as dictated by the dependency graph, compare Example 21. Finally, right hand-sides of DPs are simplified and rules are restricted to usable ones (Theorem 7 and Theorem 10).

- *Relative decomposition and orderings:* At this stage, the decomposition processor in conjunction with different complexity pairs (Theorem 4) is employed. To facilitate multiple cores, various instances, parameterised by different complexity pairs, are run in parallel. The processors are directed towards shifting leafs in the dependency graph from the strict to the weak component, using predecessor estimation (Theorem 9) to extend the set of shifted DPs. This enables removal of the oriented rules by the subsequent triggered DP simplifications.

  We remark that TCT synthesises complexity pairs by constructing suitable *polynomial* and *matrix interpretations* (of dimension one to four) that induce polynomial bounds, as well as *(small) polynomial path orders* [8, 9]. For polynomial interpretations, TCT essentially interprets constructors by *strongly linear* polynomials [3] only. For matrix interpretations our strategy uses the criterion of Middeldorp et al. [41] based on *automata techniques*.

  In the present strategy, we employ interpretations up to induced degree two, using rather low coefficients (ranging up to seven). Higher parameters show significant effect on the execution time, without improving much on the applicability of the strategy. To keep the complexity certificate tight, the application of complexity pairs is ordered by the induced degree (starting from zero).

- *DG decomposition:* Dependency graph decomposition is the central part of the strategy, it is employed whenever orderings fail to produce a result. In general, there is no favourable choices to pick a lower component $\mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp$ suitable for Theorem 11. We decided to pick for the lower component a maximal set of forward closed nodes excluding *topmost cycles* in the DG of the considered problem, i.e. cycles not reachable from any other cycle in the DG. Successive applications of DG decomposition are thus performed *outside in*.

5.1. Experimental Evaluation

In the following, we contrast the strength of TCT with its default proof-search strategy DGD to the comparable provers AProVE [42] and CaT. As AProVE features only support for innermost rewriting, the tool is excluded from benchmarks that do not restrict inputs to innermost complexity problem. Dual, CaT does not feature dedicated support for innermost rewriting and is therefore excluded on experiments on innermost problems.

To show the effect of the various methods, we (partly) contrast our default strategy DGD to the following modifications.

- *Direct:* With this strategy we check the power of direct methods. Precisely the complexity pairs outlined in stage *relative decomposition and orderings* are employed in a *direct* setting, i.e. relying on Theorem 2 only.

- *RD:* With this configuration we indicate the strength of Zankl and Korp [13] decomposition processor in combination with complexity pairs. This strategy corresponds to the stage *relative decomposition and orderings* of our default strategy DGD.

- *DP+RD:* This strategy extends the strategy RD with the weak dependency pairs and dependency tuples processor. More precise, the strategy correspond to the default strategy with stage DG decomposition disabled. Should stage *relative decomposition and orderings* fail, the computation is aborted.

- *DGD-g:* This strategy is a variation of the default strategy DGD where *DG decomposition* is employed exhaustively, immediately after initial *DP simplifications*.

Setup. All experiments were conducted on a machine with a 8 dual core AMD Opteron™ 885 processors running at 2.60GHz, and 64Gb of RAM.[9] Every test was run with a generous timeout of 300 seconds. The employed version of TCT is version 2.0, binaries for AProVE and CaT are the ones provided by the annual termination competition of 2012.[10]

Data Sets. Unarguably, the *termination problem data base* (*TPDB* for short), underlying the international termination competition, is the most extensive selection of examples available. Still, it was primarily intended as a means to assess the strength of termination provers, and falls short of interesting examples for runtime complexity analysis. We have therefore compiled two collections of TRSs specifically targeted at runtime complexity analysis.

- *Data set RaML:* To test the applicability of runtime complexity analysis of TRSs in the context of program analysis, we have employed a naive (but complexity preserving) transformation of RaML programs considered in [27] into TRSs.[11] We remark that in this transformation, we do not take typing information into account. We however use the call-by-value strategy adopted by RaML to obtain an innermost runtime complexity problem. Also, built-in operations like comparison operations on Integers and Boolean operations are assigned unitary cost. This is achieved by modeling their semantics with rewrite rules occurring in the weak component of the constructed problem. Data set RaML collects the examples obtained by this translation from the example collection available in the source repository of the RaML prototype.[12]

- *Data set RC:* This data set is mostly a compilation of examples found in the literature on the runtime complexity analysis of term rewrite systems. Also, it contains various sorting algorithms on lists and standard examples of Peano arithmetic.

Assessment. In Table 1 we contrast AProVE and TCT in the above mentioned configurations on the data set RaML. Here entries indicate the estimated upper bounds on the runtime complexity. An entry ? indicates that the tool gave up, and an entry ∞ indicates that the tool was aborted due to a timeout. The graph depicted in Figure 9 summarise the data of Table 1 for AProVE and selected configurations of TCT. Here totals on solved instances are plotted against the execution time. To put emphasis on the precision of the obtained certificate, we only take optimal solutions, in the sense that no other run of AProVE or TCT produced a tighter upper bound, into account. In contrast, dotted lines indicate the overall number of systems, disregarding optimality.

The experiments clearly illustrates the need for transformations. Only two examples could be solved by applying complexity pairs directly. The results drawn in Column RD in Table 1 shows that relative decomposition gives an improvement on the direct approach. Column DP+RD indicates the usefulness of the dependency pair transformations. Two additional problems can be solved in the dependency pair

[9] Average PassMark CPU Mark 2851; data from `http://www.cpubenchmark.net/`.
[10] Available through `http://termcomp.uibk.ac.at/`.
[11] The corresponding tool is available online at `http://cl-informatik.uibk.ac.at/cbr/tools/RaML/`.
[12] Available online at `http://raml.tcs.ifi.lmu.de/`.

	Direct	RD	DP+RD	DGD-g	DGD	AProVE
appendall	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
bfs	?	?	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	∞
bftmmult	?	?	∞	$\mathcal{O}(n^4)$	$\mathcal{O}(n^3)$	∞
bitonic	?	?	∞	$\mathcal{O}(n^4)$	$\mathcal{O}(n^4)$	∞
bitvectors	∞	∞	∞	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	∞
clevermmult	?	?	?	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
duplicates	?	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
dyade	?	?	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
eratosthenes	?	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	∞
flatten	?	?	?	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
insertionsort	?	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
listsort	?	?	?	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	∞
lcs	?	?	?	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	∞
martix	?	?	∞	$\mathcal{O}(n^4)$	$\mathcal{O}(n^3)$	∞
mergesort	?	∞	∞	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	∞
minsort	?	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$
queue	?	?	∞	$\mathcal{O}(n^5)$	$\mathcal{O}(n^5)$	∞
quicksort	?	∞	∞	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	∞
rationalpotential	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
splitandsort	?	?	?	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	∞
subtrees	?	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
tuples	?	?	∞	∞	∞	∞

Table 1: Experimental results on data set RaML.

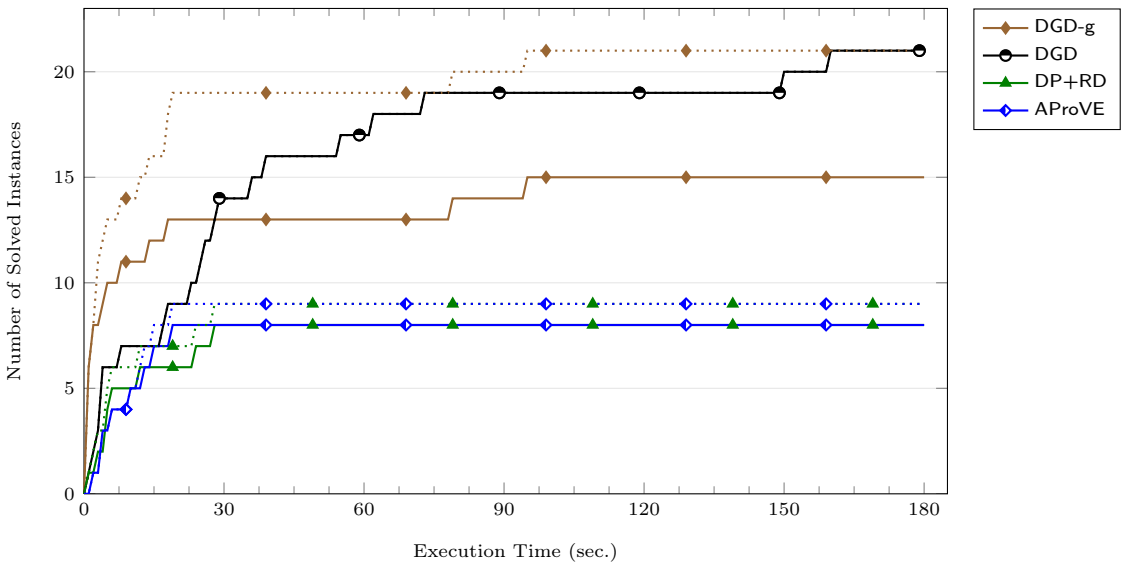


Figure 9: Data set RaML: number of solved instances, depending on execution time.

setting, and for three problems a more precise certificate could be obtained. Still, the majority of the problems remain unsolved.

The results concerning configurations DGD and DGD-g indicate the gain in power of our tool through DG decomposition. Out of the 22 examples, only the example *tuples*, which computes for fixed $i = 1, \ldots, 4$ and given set S (encoded as list) the set of all subsets of S of size i, cannot be handled. The runtime complexity of this example is perfectly described by the resource polynomials defined in terms of binomial coefficients of [27]. It is also worth mentioning that in 30 seconds, after which AProVE and configuration DP+RD fail to handle any further system, configurations DGD and DGD-g can handle significantly more systems. As we mentioned above, incompleteness is a clear drawback of dependency graph decomposition, compare also Example 24. Figure 9 indicates that even thought immediate, iterative application of this processors (as implemented in the configuration DGD-g) significantly improves the execution time, precision of the certificate is lost. In contrast, the standard configuration DGD retains precision.

Finally we remark that the RaML-prototype developed by Hoffmann et al. [21] beats TCT on the (untransformed) data set, in precision of the estimated bound, and in the speed of the analysis. We attribute this mainly to the naive transformation of RaML programs to TRSs. The RaML-prototype can use for instance domain information given by the semantics of RaML programs. This information is lost during transformation. As for the above mentioned rewriting tools, also the RaML-prototype is not capable of inferring logarithmic bounds. As a consequence, not every example was classified optimal. For instance, the runtime complexity of program *mergesort*, which implements the mergesort algorithm on list, should have complexity $\mathcal{O}(n \cdot \log(n))$, but all tests classified the runtime of this program as quadratic or above.

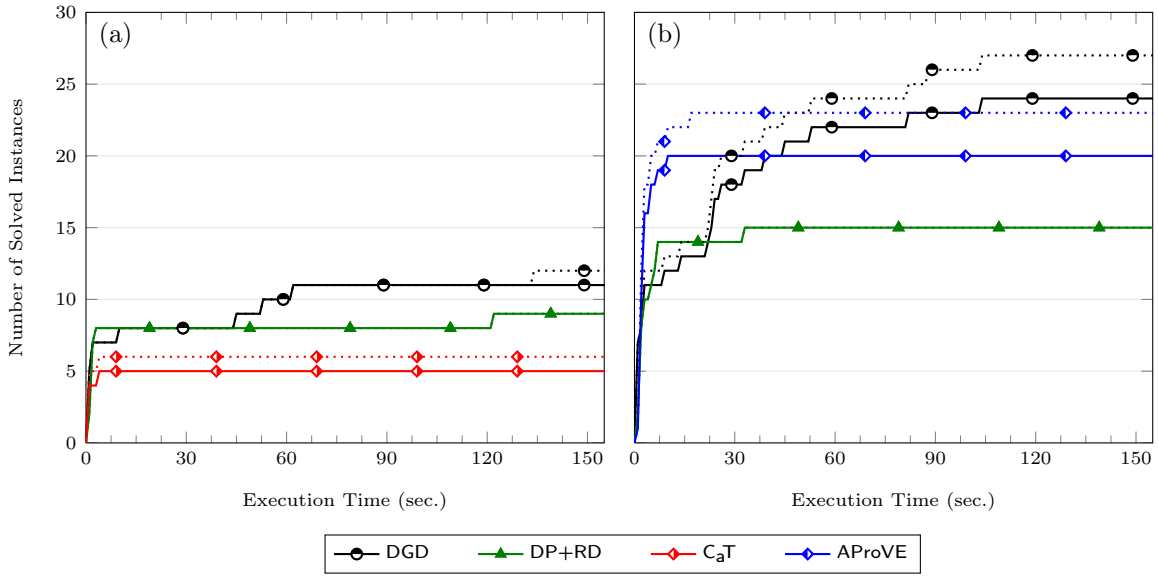


Figure 10: Data set RC for (a) full and (b) innermost rewriting: number of solved instances, depending on execution time.

In Tables 2 we compare our tool TCT to CaT (full rewriting) and AProVE (innermost rewriting) respectively, on the data set RC. For TCT we indicate results with respect to configuration without DG decomposition (configuration DP+RD) and including DG decomposition (configuration DGD). Figure 10 relates again execution times to the number of solved instances, as in Figure 9. The results are similar to the ones on the data set RaML. In contrast to innermost rewriting, TCT can only partly benefit from DG decomposition. Observe that weak dependency pairs leave rules from the input problem in the strict component. In contrast, DG decomposition operates on dependency pairs only.

	Full rewriting			Innermost rewriting		
Answer	DP+RD	DGD	CaT	DP+RD	DGD	AProVE
jones1	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
jones2	$\mathcal{O}(n)$	$\mathcal{O}(n)$	?	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
jones4	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
jones5	∞	∞	?	∞	∞	∞
jones6	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
flatten	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
reverse	∞	$\mathcal{O}(n^2)$	?	∞	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
shuffle	∞	$\mathcal{O}(n^3)$	?	∞	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$
shuffleshuffle	∞	$\mathcal{O}(n^4)$	?	∞	$\mathcal{O}(n^4)$	∞
clique	∞	∞	?	∞	$\mathcal{O}(n^3)$	?
dcquad	∞	∞	?	∞	∞	$\mathcal{O}(n^2)$
egypt	∞	∞	?	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
eratosthenes	∞	∞	?	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	∞
lcs	∞	?	∞	∞	∞	∞
mmult-nat	∞	∞	?	∞	$\mathcal{O}(n^5)$	$\mathcal{O}(n^2)$
qbf	∞	∞	?	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
sat	∞	∞	?	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
z86	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
bits	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
div	$\mathcal{O}(n)$	$\mathcal{O}(n)$	?	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
mult	∞	∞	?	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
mult2	∞	∞	?	∞	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$
quad	∞	∞	?	∞	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
square	∞	∞	?	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
bubblesort-nat	∞	∞	?	∞	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$
isort	∞	∞	?	∞	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
mergesort	∞	∞	?	∞	$\mathcal{O}(n^4)$	∞
mergesort-nat	∞	∞	?	∞	$\mathcal{O}(n^5)$	∞
quicksort-buggy	$\mathcal{O}(n)$	$\mathcal{O}(n)$	?	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
quicksort	∞	∞	?	∞	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$

Table 2: Experimental results on data set RC.

6. Conclusion

We have presented a combination framework for automated polynomial complexity analysis of term rewrite systems. The framework is general enough to reason about both runtime and derivational complexity, and to formulate a majority of the techniques available for proving polynomial complexity of rewrite systems. On the other hand, it is concrete enough to serve as a basis for a modular complexity analyser, as demonstrated by our automated complexity analyser TCT which closely implements the discussed framework.

Besides the combination framework we have introduced the notion of $\mathcal{P}$-monotone complexity pair that unifies the different orders used for complexity analysis in the cited literature, and we have introduced or adapted various syntactic simplification techniques. Last but not least, we have presented the dependency graph decomposition processor. This processor is easy to implement, and greatly improves modularity. The provided experimental results highlight the strength of our framework, and in particular of the dependency graph decomposition processor.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable and detailed comments that greatly helped in improving the presentation.

References

[1] A. Ben-Amram, Monotonicity Constraints for Termination in the Integer Domain, Logical Methods in Computer Science 7 (3).

[2] A. Ben-Amram, M. Vainer, Bounded Termination of Monotonicity-Constraint Transition Systems, CoRR abs/1202.4281.

[3] G. Bonfante, A. Cichon, J.-Y. Marion, H. Touzet, Algorithms with Polynomial Interpretation Termination Proof, Journal on Functional Programming 11 (1) (2001) 33–53.

[4] D. Hofbauer, C. Lautemann, Termination Proofs and the Length of Derivations, in: Proc. of 3rd International Conference on Rewriting Techniques and Applications, vol. 355 of *LNCS*, Springer, 167–177, 1989.

[5] N. Hirokawa, G. Moser, Automated Complexity Analysis Based on the Dependency Pair Method, in: Proc. of 4th International Joint Conference on Automated Reasoning, vol. 5195 of *LNAI*, Springer, 364–380, 2008.

[6] M. Avanzini, G. Moser, Closing the Gap Between Runtime Complexity and Polytime Computability, in: Proc. of 21st International Conference on Rewriting Techniques and Applications, vol. 6 of *LIPIcs*, Leibniz-Zentrum fuer Informatik, 33–48, 2010.

[7] U. Dal Lago, S. Martini, On Constructor Rewrite Systems and the Lambda Calculus, Logical Methods in Computer Science 8 (3) (2012) 1–27.

[8] M. Avanzini, N. Eguchi, G. Moser, New Order-theoretic Characterisation of the Polytime Computable Functions, Theoretical Computer Science To appear.

[9] M. Avanzini, G. Moser, Polynomial Path Orders, Logical Methods in Computer Science 9 (4).

[10] G. Moser, A. Schnabl, J. Waldmann, Complexity Analysis of Term Rewriting Based on Matrix and Context Dependent Interpretations, in: Proc. of 28th Conference on Foundations of Software Technology and Theoretical Computer Science, LIPIcs, Leibniz-Zentrum fuer Informatik, 304–315, 2008.

[11] L. Noschinski, F. Emmes, J. Giesl, A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems, in: Proc. of 23rd CADE, LNAI, Springer, 422–438, 2011.

[12] M. Avanzini, POP* and Semantic Labeling Using SAT, in: Interfaces: Explorations in Logic, Language and Computation, ESSLLI 2008 and ESSLLI 2009 Student Sessions. Selected Papers, vol. 6211 of *LNCS*, Springer, 155–166, 2010.

[13] H. Zankl, M. Korp, Modular Complexity Analysis via Relative Complexity, in: Proc. of 21st International Conference on Rewriting Techniques and Applications, vol. 6 of *LIPIcs*, Leibniz-Zentrum fuer Informatik, 385–400, 2010.

[14] G. Moser, Proof Theory at Work: Complexity Analysis of Term Rewrite Systems, CoRR abs/0907.5527, habilitation Thesis.

[15] N. Hirokawa, G. Moser, Complexity, Graphs, and the Dependency Pair Method, in: Proc. of 15th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning, 652–666, 2008.

[16] N. Hirokawa, G. Moser, Automated Complexity Analysis Based on the Dependency Pair Method, CoRR abs/1102.3129.

[17] M. Avanzini, G. Moser, Tyrolean Complexity Tool: Features and Usage, in: Proc. of 24th International Conference on Rewriting Techniques and Applications, vol. 21 of *LIPIcs*, Leibniz-Zentrum fuer Informatik, 71–80, 2013.

[18] N. Hirokawa, G. Moser, Automated Complexity Analysis Based on Context-Sensitive Rewriting., in: Proc. of 25th International Conference on Rewriting Techniques and Applications and the 12th International Conference on Typed Lambda Calculi and Applications, to appear., 2014.

[19] S. Gulwani, F. Zuleger, The Reachability-Bound Problem, in: Proc. of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 292–304, 2010.

[20] F. Zuleger, S. Gulwani, M. Sinn, H. Veith, Bound Analysis of Imperative Programs with the Size-Change Abstraction, in: Proc. of 18th International Symposium on Static Analysis, vol. 6887 of *LNCS*, Springer, 280–297, 2011.
[21] J. Hoffmann, K. Aehlig, M. Hofmann, Resource Aware ML, in: Proc. of 24th International Conference on Computer Aided Verification, vol. 7358 of *LNCS*, Springer, 781–786, 2012.
[22] G. Bonfante, J.-Y. Marion, R. Péchoux, Quasi-interpretation Synthesis by Decomposition and an Application to Higher-order Programs, in: Proc. of 4th International Conference on Theoretical Aspects of Computing, vol. 4711 of *LNCS*, Springer, 410–424, 2007.
[23] M. Avanzini, G. Moser, A Combination Framework for Complexity, in: Proc. of 24th International Conference on Rewriting Techniques and Applications, vol. 21 of *LIPIcs*, Leibniz-Zentrum fuer Informatik, 55–70, 2013.
[24] M. Avanzini, Verifying Polytime Computability Automatically, Ph.D. thesis, University of Innsbruck, available at `http://cl-informatik.uibk.ac.at/users/zini/publications/`, 2013.
[25] C. Choppy, S. Kaplan, M. Soria, Complexity Analysis of Term-Rewriting Systems, Theoretical Computer Science 67 (2–3) (1989) 261–282.
[26] S. Bellantoni, S. Cook, A new Recursion-Theoretic Characterization of the Polytime Functions, Computational Complexity 2 (2) (1992) 97–110.
[27] J. Hoffmann, K. Aehlig, M. Hofmann, Multivariate Amortized Resource Analysis, in: Proc. of 38th Annual Symposium on Principles of Programming Languages, ACM, 357–370, 2011.
[28] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. Ramírez, G. Román, D. Zanardini, Termination and Cost Analysis with COSTA and its User Interfaces, Electronic Notes in Theoretical Computer Science 258 (1) (2009) 109–121.
[29] C. Alias, A. Darte, P. Feautrier, L. Gonnord, Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs, in: Proc. 17th SAS, vol. 6337 of *LNCS*, 117–133, 2010.
[30] M. Hofmann, D. Rodriguez, Automatic Type Inference for Amortised Heap-Space Analysis, in: Proc. of 22nd European Symposium on Programming, vol. 7792 of *LNCS*, Springer, 593–613, 2013.
[31] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.
[32] R. Thiemann, The DP Framework for Proving Termination of Term Rewriting, Ph.D. thesis, University of Aachen, available as Technical Report AIB-2007-17, 2007.
[33] M. Korp, C. Sternagel, H. Zankl, A. Middeldorp, Tyrolean Termination Tool 2, in: Proc. of 20th International Conference on Rewriting Techniques and Applications, vol. 5595 of *LNCS*, Springer, 295–304, 2009.
[34] L. Noschinski, F. Emmes, J. Giesl, Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs, Journal of Automated Reasoning 51 (1) (2013) 27–56.
[35] J. Giesl, T. Arts, E. Ohlebusch, Modular Termination Proofs for Rewriting Using Dependency Pairs, Journal of Symbolic Computation 34 (2002) 21–58.
[36] T. Arts, J. Giesl, Termination of Term Rewriting using Dependency Pairs, Theoretical Computer Science 236 (1–2) (2000) 133–178.
[37] G. Moser, A. Schnabl, The Derivational Complexity Induced by the Dependency Pair Method, Logical Methods in Computer Science 7 (3).
[38] N. Hirokawa, A. Middeldorp, Dependency Pairs Revisited, in: Proc. of 15th International Conference on Rewriting Techniques and Applications, vol. 3091 of *LNCS*, Springer, 249–268, 2004.
[39] C. S. Lee, N. D. Jones, A. M. Ben-Amram, The Size-change Principle for Program Termination, in: Proc. of 28th Annual Symposium on Principles of Programming Languages, ACM, 81–92, 2001.
[40] N. Hirokawa, A. Middeldorp, H. Zankl, Uncurrying for Termination and Complexity, Journal of Automated Reasoning 50 (3) (2013) 279–315.
[41] A. Middeldorp, G. Moser, F. Neurauter, J. Waldmann, H. Zankl, Joint Spectral Radius Theory for Automated Complexity Analysis of Rewrite Systems, in: Proc. of 4th International Conference on Algebraic Informatics, vol. 6742 of *LNCS*, Springer, 1–20, 2011.
[42] J. Giesl, P. Schneider-Kamp, R. Thiemann, Automatic Termination Proofs in the Dependency Pair Framework, in: Proc. of 3rd International Joint Conference on Automated Reasoning, vol. 4130 of *LNCS*, Springer, 281–286, 2006.