Information and Computation 248 (2016) 22–55

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

A combination framework for complexity ☆

Martin Avanzini *, Georg Moser

Institute of Computer Science, University of Innsbruck, Austria

ARTICLE INFO

Article history:
Received 15 November 2013
Available online 6 January 2016

Keywords:
Term rewriting
Resource analysis
Runtime complexity
Automation

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.

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], 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

☆ This work is partly supported by FWF (Austrian Science Fund) project I-603-N18.
* Corresponding author.
E-mail addresses: martin.avanzini@uibk.ac.at (M. Avanzini), georg.moser@uibk.ac.at (G. Moser).

http://dx.doi.org/10.1016/j.ic.2015.12.007
0890-5401/

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 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 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.

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