ISR 2020 Advanced Course Proposal

Automated Complexity Analysis for Term Rewriting

Carsten Fuhs

Abstract

Complexity analysis for term rewriting aims to infer bounds on the length of the longest rewrite sequence as a function of the size of the considered start terms. In this course, we look into static analysis techniques that can be used as ingredients for automatic complexity analysis tools for term rewriting. Such tools take as input a term rewrite system and provide as output asymptotic upper or lower worst-case complexity bounds (e.g., $O(n^2)$) for this term rewrite system.

1 Outline

Complexity analysis for term rewrite systems (TRSs) investigates the length of the longest rewrite sequence (or: longest derivation) as a function of the size of its start term. If the set of start terms is not restricted, this complexity function is known as the derivational complexity of the TRS [11]. It is in general not computable; that is why the development of techniques and tools has focused on sufficient criteria for the inference of (not necessarily tight) upper and lower asymptotic bounds of the complexity function.

Research in this area initially dealt with upper bounds on the derivational complexity induced by termination proof techniques: if termination of a given TRS is proved using a particular technique, this implies that the derivational complexity of the TRS cannot exceed a bound specific to the proof technique and its parameters [11].

However, from a program analysis perspective, derivational complexity is not a suitable complexity measure: even a simple TRS with the two rules $\text{double}(0) \rightarrow 0$ and $\text{double}(s(x)) \rightarrow s(s(\text{double}(x)))$ to double a natural number has exponential derivational complexity. The issue is that derivational complexity considers start terms of arbitrary shape; in particular, start terms with nested defined symbols like $\text{double}(\text{double}(\ldots \text{double}(s(0))) \ldots))$ are allowed and may cause exponential derivational complexity. In contrast, computing the double of a natural number $\text{double}(s(\ldots s(0) \ldots))$ (intuitively: applying the function $\text{double}$ to data) has linear complexity. This latter notion, where defined symbols in the start term are allowed only at its root, is significantly closer to complexity analysis for conventional programming languages. The corresponding complexity function is known as the runtime complexity [10] of a TRS.

In this course, we will discuss a selection of automated techniques for inferring asymptotic lower and upper bounds on derivational and runtime complexity of TRSs. For upper bounds, the dependency pair method from termination analysis [1] has been adapted to allow for a certain modularity in the analysis [10, 14, 3]. Further techniques to find upper bounds include transformations between different complexity problems for rewriting [5, 7] or from rewriting to programs on integer numbers [13]. As witnesses for lower bounds, one can find “decreasing loops” or identify a family of start terms $t_n$, parametric in the term size $n$ for which one can prove inductively that corresponding rewrite sequences of a certain length exist [6].

If time permits, we may also briefly look at the application of complexity analysis tools for term rewriting as backends for the complexity analysis of programming languages such as Prolog [9], OCaml [2], and Java [12].
2 Duration and Exercises

The course is designed for 2 slots of 90 minutes each. These will also include selected exercises and the possibility to work with existing complexity analysis tools such as AProVE \cite{Giesl2017} and TcT \cite{Avanzini2016}.

References


