Automatic Complexity Analysis of Programs

Florian Frohn and Jürgen Giesl*

LuFG Informatik 2, RWTH Aachen University

1 Introduction

Automated complexity analysis has become an increasingly important subject. During the last years, we improved the power of our tool AProVE [5] w.r.t. complexity analysis significantly. The keys to these improvements are techniques to prove *lower bounds* on the worst-case complexity [2, 3], cf. Sect. 2, and *transformational techniques* [1,4,6] which allow us to reuse existing methods for worst-case upper bounds for various input languages, cf. Sect. 3.

2 Proving Worst-Case Lower Bounds

Our first technique to prove worst-case lower bounds, the so-called *induction technique* [2], operates on term rewrite systems (TRSs). It starts with generalizing sample rewrite sequences automatically, resulting in *conjectures*, i.e., finite representations of infinite families of rewrite sequences. Then, it proves the validity of conjectures by induction, resulting in *lemmas*. The structure of the induction proofs finally gives rise to a lower bound on the complexity of each lemma. Afterwards these lemmas can be used to speculate and prove further conjectures.

In [2], we also introduced *loop detection*, a syntactic criterion to prove linear and exponential lower bounds. It searches for *decreasing loops*, a generalization of the well-known notion of *loops* which are witnesses for non-termination of TRSs. The main result is that every TRS with a decreasing loop has at least linear, and every TRS with multiple compatible decreasing loops has at least exponential complexity.

Since TRSs do not have built-in integers, we also introduced a technique to prove worst-case lower bounds for integer transition systems (ITSs) [3]. It simplifies programs with complicated control flow via *loop acceleration*, i.e., loops are replaced with cost-annotated straight-line code which has the effect of several loop iterations. The resulting loop-free programs are suitable to deduce lower bounds automatically.

* Supported by the DFG grant GI 274/6-1 and the Air Force Research Laboratory (AFRL).

3 Transformational Techniques for Upper Bounds

Until recently, AProVE could only analyze the complexity of TRSs assuming an eager (innermost) evaluation strategy. In order to analyze the complexity of full rewriting (i.e., assuming a completely unrestricted evaluation strategy), we developed a sufficient criterion to prove that innermost evaluation is the least efficient strategy for a given TRS [4]. If it applies, then we can safely use existing techniques for innermost rewriting to analyze the complexity of full rewriting, since we know that the worst-case complexity of the TRS is captured by innermost rewriting.

To benefit from recent improvements of complexity analysis techniques for ITSs, we furthermore developed a complexity-preserving transformation from TRSs to *recursive ITSs* (RITSs, an extension of ITSs that allows arbitrary recursion) which abstracts data structures to their size [6].

Since many tools for ITSs do not support (non-tail) recursion, we also introduced an approach to analyze RITSs via techniques and tools for standard ITSs [6]. It analyzes the runtime and the size of the result of non-recursive program parts independently and abstracts from calls to already analyzed program parts using the obtained runtime- and size-bounds.

Finally, in [1] we showed that the state of the art for automated complexity analysis of Java programs can be improved significantly by abstracting objects to integers using a novel size measure. Afterwards, existing tools are used to analyze the complexity of the resulting ITSs. In contrast to established measures like path-length, our measure also takes the size of elements of data structures into account.

4 Conclusion

We discussed novel techniques to infer worst-case lower bounds as well as transformational techniques to reuse existing approaches for worst-case upper bounds for different input languages. An example for an application of such techniques is the DARPA STAC project, [1] where AProVE is used to find or to prove the absence of denial of service and timing vulnerabilities in Java programs. [2]

References

[1] F. Frohn and J. Giesl. Complexity Analysis for Java with AProVE. In *Proc. iFM '17*, 2017. To appear.

[2] F. Frohn, J. Giesl, J. Hensel, C. Aschermann, and T. Ströder. Lower bounds for runtime complexity of term rewriting. *Journal of Automated Reasoning*, 59(1):121–163, 2017.

[3] F. Frohn, M. Naaf, J. Hensel, M. Brockschmidt, and J. Giesl. Lower runtime bounds for integer programs. In *Proc. IJCAR '16*, LNAI 9706, pages 550–567, 2016.

[4] F. Frohn and J. Giesl. Analyzing runtime complexity via innermost runtime complexity. In *Proc. LPAR '17*, EPiC 46, pages 249–268, 2017.

[5] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

[6] M. Naaf, F. Frohn, M. Brockschmidt, C. Fuhs, and J. Giesl. Complexity analysis for term rewriting by integer transition systems. *Proc. FroCoS '17*, 2017. To appear.

[1] http://www.darpa.mil/program/space-time-analysis-for-cybersecurity

[2] http://www.draper.com/news/draper-s-cage-could-spot-code-vulnerable-denial-service-attacks