Analyzing Runtime and Size Complexity of Integer Programs (Abstract)

Marc Brockschmidt[1], Fabian Emmes[1], Stephan Falke[2], Carsten Fuhs[3], and Jürgen Giesl[1]

1 **RWTH Aachen University, Germany**
2 **Karlsruhe Institute of Technology, Germany**
3 **University College London, UK**

Abstract

We developed a modular approach to automatic complexity analysis. Based on a novel alternation between finding symbolic time bounds for program parts and using these to infer size bounds on program variables, we can restrict each analysis step to a small program part while maintaining a high level of precision. Extensive experiments with the implementation of our method demonstrate its performance and power in comparison with other tools. In particular, our method finds bounds for many programs whose complexity could not be analyzed by automatic tools before.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.2.8 Metrics, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Symbolic Complexity Bounds, Termination Analysis, Runtime Complexity, Size Complexity, Integer Programs

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

Methods for automatically proving termination of imperative programs have received increased attention recently. But in many cases, termination is not sufficient. Instead, the program should terminate in reasonable (typically, (pseudo-)polynomial) time. We build upon the well-known observation that (e.g., polynomial) rank functions used for termination proofs implicitly also provide a runtime complexity bound. However, this only holds for proofs using a *single* rank function. In practice, larger programs are usually handled by a disjunctive or lexicographic combination of simple rank functions. Deriving a complexity bound in this setting is much harder, as the two examples below illustrate.

For both programs, the lexicographic rank function $\langle f_1, f_2 \rangle$ proves termination, where f_1 measures states by $\mathtt{i}$ and f_2 is $\mathtt{x}$. However, the program on the left has linear runtime, while the program on the right has quadratic complexity. The difference between the programs is in the *size* of $\mathtt{x}$ after the first loop. To handle such effects, our method derives *runtime complexity* bounds for parts of the program and uses them to deduce *size complexity* bounds for program variables at certain locations. We measure the *size* of integers by their absolute values. These size bounds allow to derive more runtime complexity bounds, and the process continues until all loops and variables have been handled.

```
while i > 0 do
  i = i − 1
done
while x > 0 do
  x = x − 1
done
```

```
while i > 0 do
  i = i − 1
  x = x + i
done
while x > 0 do
  x = x − 1
done
```

For the example on the right, our method first proves that the first loop is executed linearly often using the rank function $\mathtt{i}$. Then, it deduces that $\mathtt{i}$ is bounded by its initial value $\mathtt{i}_0$ in all loop iterations. Combining these observations, our approach infers that $\mathtt{x}$ is incremented by a value bounded by $\mathtt{i}_0$ at most $\mathtt{i}_0$ times, yielding that $\mathtt{x}$ is bounded by $\mathtt{x}_0 + \mathtt{i}_0^2$. Finally, our method detects that the second loop is executed $\mathtt{x}$ times, and combines this with our bound $\mathtt{x}_0 + \mathtt{i}_0^2$ on the value of $\mathtt{x}$ when entering the second loop. This allows us to conclude that the program's runtime is bounded by $\mathtt{i}_0 + \mathtt{i}_0^2 + \mathtt{x}_0$.

Our main contribution is a novel approach which *alternates* between finding *runtime bounds* and finding *size bounds* for sequential imperative programs operating on integer data with (potentially non-linear) arithmetic and (unbounded) non-determinism. We apply this general approach to obtain two main results:

1. A novel method to deduce (often non-linear) *size bounds* on program variables by combining bounds for local variable changes with runtime bounds.
2. A new *modular* method to compute symbolic *runtime bounds* for isolated program parts. These runtime bounds are based on size bounds for variables that may have been modified in the preceding parts of the program. In this way, we only need to consider small program parts at a time, allowing our approach to *scale* to larger programs.

Several methods to determine symbolic runtime complexity bounds have been developed in recent years, e.g., [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. To the best of our knowledge, our approach is the first that implements a combined, alternating size and runtime analysis for imperative programs. To evaluate our method, we have created a simple prototype `KoAT`.

We compared this prototype with the complexity analyzers `PUBS` [1, 2] and `Rank` [3] on 682 examples from the literature on termination and complexity analysis of integer programs. The table on the side illustrates how often the tools could infer a runtime bound for the example set. Here 1, $\log n$, n, $n \log n$, n^2, n^3, and $n^{>3}$ represent their corresponding asymptotic classes and EXP is the class of exponential functions. The column "Time" gives the average runtime on those examples where the respective tool was successful. The table shows that on this collection, our approach was substantially more powerful than the two other previous tools and still efficient.

	1	$\log n$	n	$n \log n$	n^2	n^3	$n^{>3}$	EXP	Time
`KoAT`	102	0	151	0	58	3	3	0	1.7 s
`PUBS`	85	4	104	1	13	4	0	6	.3 s
`Rank`	56	0	19	0	8	1	0	0	.5 s

References

1 E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *JAR*, 46(2):161–203, 2011.

2 E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *TCS*, 413(1):142–159, 2012.

3 C. Alias, A. Darte, P. Feautrier, L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS '10*, pages 117–133, 2010.

4 M. Avanzini and G. Moser. A combination framework for complexity. In *RTA '13*, pages 55–70, 2013.

5 R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. `ABC`: Algebraic bound computation for loops. In *LPAR-16*, pages 103–118, 2010.

6 J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. Symbolic evaluation graphs and term rewriting: A general methodology for analyzing logic programs. In *PPDP '12*, pages 1–12, 2012.

7 S. Gulwani, K. K. Mehra, and T. M. Chilimbi. `SPEED`: Precise and efficient static estimation of program computational complexity. In *POPL '09*, pages 127–139, 2009.

8 J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *TOPLAS*, 34(3), 2012.

9 J. A. Navas, E. Mera, P. López-García, and M. V. Hermenegildo. User-definable resource bounds analysis for logic programs. In *ICLP '07*, pages 348–363, 2007.

10 L. Noschinski, F. Emmes, and J. Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *JAR*, 51(1):27–56, 2013.

11 F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS '11*, pages 280–297, 2011.