ComplexityParser: an automatic tool for certifying poly-time complexity of Java programs

Emmanuel Hainry, Emmanuel Jeandel, Romain Péchoux, and Olivier Zeyen

Project Mocqua, CNRS, Inria, and Université de Lorraine, LORIA, Nancy, France

Abstract. ComplexityParser is a static complexity analyzer for Java programs providing the first implementation of a tier-based typing discipline. The input is a file containing Java classes. If the main method can be typed and, provided the program terminates, then the program is guaranteed to do so in polynomial time and hence also to have heap and stack sizes polynomially bounded. The application uses antlr to generate a parse tree on which it performs an efficient type inference: linear in the input size, provided that the method arity is bounded by some constant.

1 Introduction

Motivations. The use of tiering techniques to certify program complexity was kick-started by the seminal works of Bellantoni-Cook [4] and Leivant-Marion [22], that provide sound and complete characterizations of the class of functions computable in polynomial time `FP`. Tiering was later adapted to several other complexity classes such as `FPSPACE` [23], `NC` [8, 28, 21], or `L` [28].

Despite these numerous theoretical results, tiering, until now, had no practical application in automatic complexity analysis because of a lack of expressive power. Indeed, the tiering discipline severely constrains the way first order functional programs can be written. This problem has been solved by the cornerstone work of [25] that has exhibited and studied the relations between tiering and non-interference on imperative programs. It has been extended to fork processes [13], object-oriented programs [24, 14, 15], and type-2 polynomial time [12], hence showing its portability and paving the way for practical implementations.

The core idea is the design of a type discipline ensuring polynomial time termination. The type system, inspired by non-interference, splits program variables, expressions, and statements in two disjoint *tiers*, denoted **0** and **1**, and enforces constraints on the data flow, the control flow and sizes:

- *Data flow.* Data flows from tier **0** to tier **1** are prohibited, *e.g.* if `x` is of tier **0** and `y` is of tier **1** then the statement `y = x;` cannot be typed, and data flows from tier **1** to tier **0** are prevented for non-primitive data. It forbids tier **1** data from increasing by side effects.
- *Control flow.* Loop control flows depending on tier **0** data are forbidden. *I.e.* in a `while(e){c}` statement, the expression `e` is enforced to be of tier **1**. This implies that all the variables of `e` are of tier **1**.

– *Size control.* Tier **1** data cannot make the memory size increase, *e.g.* if `x` is of tier **1** then `x = x-1`; is typable but `x = x+1`; and `x = new C(x)`; are not typable.

Theorem 1 ([15]). *If a program is typable and terminates on input d then its runtime, heap space, and stack space are polynomially bounded in the size of d.*

Contribution. This paper describes the architecture of ComplexityParser, a tier-based automatic cost analyzer for Java programs implementing the type system of [14, 15] in Java, built using Maven, and whose source code is fully available at `https://gitlab.inria.fr/hainry/complexityparser`. The application receives as input a text file program and infers a type (called tier) as output. If the type inference succeeds and the analyzed program terminates then: the program is certified to have a worst case execution time bounded polynomially, by Theorem 1. Even in case the full program cannot be typed, the success of typing some methods will be indicated, giving a complexity certificate for those methods provided they terminate.

ComplexityParser is, to our knowledge, the first implementation of a tier-based technique to a realistic programming language: the analysis can deal with an expressive fragment of Java programs with while loops, recursive methods, exception handlers, and inheritance (including overriding methods) and allowing the programmer to define inductive and cyclic data (List and Ring for example). This application is, with RAJA [18, 19] and RaML [17], one of the first practical type-based approaches for certifying program complexity. It provides the first implementation of a complete technique with respect to polynomial time. Its code is open source (Apache 2.0 license), and its performance is good: the type analysis is linear in the size of the input program in practice. Moreover, it is fully automatic as types (tiers) are inferred and do not need to be explicitly provided by the programmer.

The implemented analyses [25, 14, 15] are complete for the class of polynomial time computable functions, *i.e.* capture all functions. However they can yield false negatives, *i.e.* programs running in polynomial time that cannot pass the type inference, which cannot be avoided as the problem of knowing if a program computes a polytime function is not decidable [16].

Related works. We know of few alternative tools studying the complexity of Object Oriented programs.

– AProVE [9], initially a termination tool, was expanded to treat complexity by translating constraints into integer constraints, then using an external integer programming solver. AProVE works on Java bytecode and supports most features of the language except recursion. It outputs $O(n^k)$ bounds.
– costa [1, 2] infers symbolic costs upper bounds on Java bytecode by generating recurrence equations relatively to a chosen cost measure and then by finding an upper bound on a solution to these equations. The upper bounds it computes are combinations of polynomials and logarithms.

– RAJA [19] automatically computes linear bounds on the heap space used by a Featherweight Java program. It translates the memory constraints into graphs and solves linear cases. Exceptions, overloading and variable shadowing are also not treated. Note that this tool captures LinSpace.
– SPEED [10, 11] infers symbolic complexity bounds on the number of statements executed by a `C++` program. It relies on integer counters inserted in the program by the programmer. As such, it is not fully automatic and does not treat objects as first class citizens.
– TcT [3, 27] is a generic complexity tool which, for what interests us analyzes Jinja bytecode. It translates code into term rewriting systems for which it uses integer programming techniques. Jinja bytecode does not support recursion and cyclic data but otherwise has all language features we checked.

A summary of the above tools' features is presented in the following table. For each tool, we check language features support (recursion, exceptions, inheritance, cyclic data), whether it gives explicit bounds, whether it works on a High-Level language (HLL) or on bytecode, and last if it is fully automatic.

Tool	Recursion	Exceptions	Inheritance	Cyclic	Bounds	HLL	Auto
ComplexityParser	Yes	Yes	Yes	Yes	No	Yes	Yes
AProVE	No	Yes	Yes	Yes	Yes	No	Yes
COSTA	Yes	Yes	Yes	No	Yes	No	Yes
RAJA	Yes	No	Partial	Yes	Yes	Yes	Yes
SPEED	Yes	No	No	No	Yes	Yes	No
TcT	No	Yes	Yes	No	Yes	No	Yes

2 ComplexityParser Overview

ComplexityParser concretizes the principles presented in Section 1. It is implemented in Java with around 5000 lines. To illustrate the analyzed language, a reduced example is given in Figure 1. More examples are available in the repository[1].

Example 1. Figure 1 implements a BList class (representing binary numbers as lists of booleans). It is annotated with the tiers inferred by the type system. For methods, the tier is of the form $\alpha_0 \times \cdots \times \alpha_n \to \alpha$ to denote that the current object is of tier α_0, the n arguments have tiers α_i, for $i \geq 1$, and the output has tier α.

– Getters have two possible tier signatures: $\mathbf{0} \to \mathbf{0}$ or $\mathbf{1} \to \mathbf{1}$, illustrating the fact that if the current object is of tier α, its fields are of tier α;
– `setTail` modifies a field, hence current object and argument must have the same tier. Since it can make the current object grow, this tier must be $\mathbf{0}$;

[1] `https://gitlab.inria.fr/hainry/complexityparser/-/tree/master/examples`

```
class BList {
    boolean value;
    BList tail;

    BList(boolean v, BList q) { value = v; tail = q; }

    boolean getValue() { return value; }    // 0 → 0 or 1 → 1
    void setTail(BList t) { tail = t; }     // 0 × 0 → 0
    BList getTail() { return tail; }        // 0 → 0 or 1 → 1
    void concat(BList other) { /* ... */ }  // 1 × 0 → 1
    int length() { /* recursive method */ } // 1 → 0
    boolean isEqual(BList other) { /* simple while loop */}
    // isEqual: 1 × 1 → 1
    boolean lessOrEqual(BList other) { /* while loop */ }
    // lessOrEqual: 1 × 1 → 1
}

class Exe {
    void main(String[] args) {
        #init
            BList b1 = new BList(true, null);
            b1 = new BList(false, b1);
            b1 = new BList(true, b1);
            BList b2 = new BList(true, null);
            b2 = new BList(false, b2);
            // b1: 1; b2: 1
        #init

        BList tail1 = b1.clone();       // tail1: 0
        tail1 = tail1.getTail();        // tail1: 0
        boolean res = b1.lessOrEqual(b2); // res: 1
        res = b1.isEqual(b2);             // res: 1
        b1.concat(b2.clone());            // OK
        res = tail1.isEqual(b1);
        // Fails as isEqual needs tail1 to be of tier 1
    }
}
```

Fig. 1. Binary numbers as lists (extract from `example11.txt`)

– `length` is recursive hence input tiers must be **1**;
– `isEqual` iterates on the current object and on `other` they must be of tier **1**.
– Variables `b1`, `b2` created inside the `#init` block automatically get tier **1**;
– The next lines of method `main` check, except the last one (line 35) which tries to apply `isEqual` on an object of tier **0** instead of tier **1**.

ComplexityParser's behavior is described by the state diagram of Figure 2. We will describe this behaviour in detail in the rest of this section.

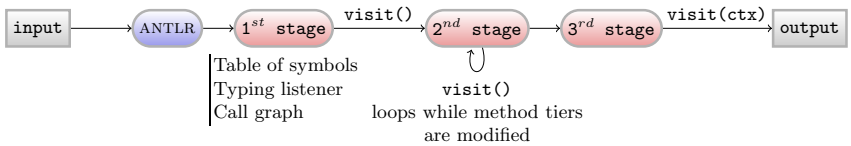

Fig. 2. Program state diagram

Input and parse tree generation. The `input` is a string encoding of a text file containing a collection of Java classes. As illustrated in Figure 1, some statements in the `main` method can be encompassed between `#init` tags to be treated as the input data of the complexity analysis. No tier-based complexity analysis is performed on these statements and, consequently, they are not subject to the typing constraints described in Section 1. While this block is necessarily fixed in our textual examples, it should be understood as a modifiable block: it is the way for the user to build the input on which the program should run. We chose this somewhat clunky way of representing input instead of simply passing them as arguments of the `main` method because Java only allows `String` arrays as arguments and translating such inputs would blur the point. The program uses ANTLR[2], a lexer/parser generator, and creates a new parser instance based on the full Java grammar described in [29] extended with an input tag `#init`, a declassification tag `#declass`, and variable tier declarations (e.g. `int<0> x;`).

Stages of type inference. Tiers and typing judgments of [15] are implemented in an object hierarchy. The class `Tier` contains the constants `T0`, `T1` and `None` for representing the tiers **0**, **1**, and "undefined", plus some useful static methods on tiers. A method can have several types that will depend on its caller context (encoded in an environment tier field `env`). When typing a method, the tiers of the `this` object and of input arguments should be mapped to the tier of the output.

Type inference is performed in 3 stages (see Figure 2) to compute the admissible method tiers and finally visiting the parse tree to check if the whole program types. The methods' admissible tier list, which is initialized as all the

[2] `www.antlr.org`

combinatorial possible tiers for each input and output in the 1st stage, is progressively reduced to encompass only the set of admissible tiers in the 2nd stage. The 3rd stage checks for admissible constructor tiers.

```
    public Object visitStatement_while(... ctx) {
        visitChildren(ctx);
        Tier t = getTier(ctx.expression());
        Tier env = getEnvironmentTier(ctx.expression());
        incrementWhileCount();
        Tier res;
        if (getRecursiveCallsCount() != 0) {
            res = Tier.None;
            env = Tier.None;
        }
        if (t == Tier.T1 && env != Tier.None){
            Tier st = getTier(ctx.statement());
            if (st == Tier.None || st == null) {
                res = Tier.None;
            } else { res = Tier.T1;}
        } else { res = Tier.None;}
        putTier(ctx, res, Tier.T1);
        return res;
    }
```

Fig. 3. Visitor method for `while` statements in `BaseStatementVisitor`

The typing discipline is performed using visitors that are specialized for each node type of the parse tree. For example, the code of the `while` statement visitor is provided in Figure 3. It implements the following rule (rephrased rule (Wh) from [15]) enforcing any while statement to be controlled by a tier `T1` expression:

$$\frac{\Gamma \vdash_{\texttt{env}} \texttt{ctx.expression()} : \texttt{T1} \quad \Gamma \vdash \texttt{ctx.statement()} : \texttt{st} \quad \texttt{env},\ \texttt{st} \in \{\texttt{T0}, \texttt{T1}\}}{\Gamma \vdash \texttt{while(ctx.expression()\{ctx.statement()\}} : \texttt{T1}}$$

Output. The `output` consists in the GUI displaying the "`Final result`", the tier of `main` method body. **0** and **1** represent *success* and entail, by Theorem 1, that if the program terminates then its runtime is polynomial. The final result can also be `None`, indicating a failure to type. This latter case does not imply that the main method complexity is not polynomial as there are false negatives. However some other method declarations in the program may have typed, which guarantees their complexity to be polytime under the termination assumption.

Complexity. Type inference for tier-based typing disciplines was shown to be linear in the size of the input program using a reduction to 2-SAT [13, 15]. However ComplexityParser's inference algorithm is based on a brute force implementation that remains linear in practice but is exponential in theory.

Let $|p|$ be the size (*i.e.* number of symbols) of the Java program given as input and $ar(p)$ be the maximal number of parameters of a method in the program p.

The first part of our algorithm builds the parse tree of the considered program using the ANTLR *ALL(*)* algorithm that is in $O(|p|^4)$ in theory [30] and produces a parse tree of size linear in the size of the input program.

Then, the 3 stages of Figure 2 visit the parse tree nodes linearly often. In the 1st stage, when visiting a method declaration node, all admissible tier combinations are added to the typing environment, costing $O(2^n)$, for a method with n parameters. Searching fixpoints in the 2nd stage has a WCET in $O(2^{ar(p)}|p|)$: at each step, either the list of possible tier combinations is decremented or the environment is unchanged and the program jumps to the 3rd stage. Typing recursive methods relies on computing the strongly connected components of a graph using the standard Kosaraju DFS algorithm [32] which also has linear complexity. The 3rd stage performs in $O(|p|)$ as it simply type-checks constructor bodies.

Putting all together, the complexity of the analysis is in $O(2^{ar(p)}|p| + |p|^4)$. We argue that $ar(p)$ should be a small constant independent of the program. Indeed, following [7], it is a good practice to *"avoid long parameter lists. As a rule, three parameters should be viewed as a maximum, and fewer is better."* The $O(|p|^4)$ part is due to ANTLR's complexity which, according to its authors, "*performs linearly on grammars used in practice*", such as the Java grammar. Finally, we claim that the whole analysis also has linear complexity in practice, as highlighted by our tests on sample programs.

3 Conclusion

ComplexityParser is an efficient and automatic high level complexity analyzer for Java programs. The analysis can deal with most well-known Java constructs. There are still some syntactic restrictions on the expressive power of the analysis such as not treating `for` loops. Plans for future work include the treatment of threads as in [26], and symbolic computation of the polynomial bounds as in [15] on execution time, heap space, and stack space. Unfortunately, due to object data controlling the flow, tight bounds are difficult to infer. Our approach requires a termination certificate (see Theorem 1). This issue can be overcome by combining our tool with existing termination techniques, *e.g.* [20, 31] or loop invariant generation techniques, *e.g.* [5, 6, 33]. We also consider interfacing with existing termination analyzers working on Java such as Julia [34], or other complexity analyzers that prove termination like AProVE or COSTA.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of Java bytecode. In: European Symposium on Programming (ESOP) 2007. pp. 157–172 (2007)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theoretical Computer Science **413**(1), 142–159 (2012)

3. Avanzini, M., Moser, G., Schaper, M.: TcT: Tyrolean complexity tool. In: Tools and Algorithms for Construction and Analysis of Systems (TACAS) 2016, pp. 407–423. Springer (2016)
4. Bellantoni, S., Cook, S.A.: A new recursion-theoretic characterization of the polytime functions. Computational Complexity **2**, 97–110 (1992)
5. Ben-Amram, A.M., Genaim, S.: On the linear ranking problem for integer linear-constraint loops. In: Principles of Programming Languages (POPL) 2013. pp. 51–62 (2013)
6. Ben-Amram, A.M., Hamilton, G.W.: Tight worst-case bounds for polynomial loop programs. In: Foundations of Software Science and Computation Structure (FoSSaCS) 2019. pp. 80–97 (2019)
7. Bloch, J.J.: Effective Java, 2nd Edition. The Java series ... from the source, Addison-Wesley (2008)
8. Bonfante, G., Kahle, R., Marion, J.Y., Oitavem, I.: Two function algebras defining functions in NC^k boolean circuits. Inf. Comput. **248**, 82–103 (2016)
9. Frohn, F., Giesl, J.: Complexity analysis for java with AProVE. In: Integrated Formal Methods. pp. 85–101. Springer (2017)
10. Gulwani, S.: SPEED: symbolic complexity bound analysis. In: Computer Aided Verification (CAV) 2009. pp. 51–62 (2009)
11. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: precise and efficient static estimation of program computational complexity. In: Principles of Programming Languages (POPL) 2009. pp. 127–139 (2009)
12. Hainry, E., Kapron, B.M., Marion, J.Y., Péchoux, R.: A tier-based typed programming language characterizing feasible functionals. In: Logic in Computer Science (LICS) 2020. pp. 535–549 (2020)
13. Hainry, E., Marion, J.Y., Péchoux, R.: Type-based complexity analysis for fork processes. In: Foundations of Software Science and Computation Structure (FoSSaCS) 2013. pp. 305–320. Springer (2013)
14. Hainry, E., Péchoux, R.: Objects in polynomial time. In: APLAS 2015. pp. 387–404. Springer (2015)
15. Hainry, E., Péchoux, R.: A type-based complexity analysis of object oriented programs. Information and Computation **261**, 78–115 (2018)
16. Hájek, P.: Arithmetical hierarchy and complexity of computation. Theoretical Computer Science **8**, 227–237 (1979)
17. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource Aware ML. In: Computer Aided Verification (CAV) 2012. LNCS, vol. 7358, pp. 781–786. Springer (2012)
18. Hofmann, M., Rodriguez, D.: Efficient type-checking for amortised heap-space analysis. In: Computer Science Logic (CSL) 2009, pp. 317–331. Springer (2009)
19. Hofmann, M., Rodriguez, D.: Automatic type inference for amortised heap-space analysis. In: Programming Languages and Systems, pp. 593–613. Springer (2013)
20. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Principles of Programming Languages (POPL) 2001. pp. 81–92 (2001)
21. Leivant, D.: A characterization of `NC` by tree recurrence. In: Foundations of Computer Science 1998. pp. 716–724. IEEE (1998)
22. Leivant, D., Marion, J.Y.: Lambda calculus characterizations of poly-time. In: Typed Lambda Calculi and Applications 1993. pp. 274–288. Springer (1993)
23. Leivant, D., Marion, J.Y.: Ramified recurrence and computational complexity II: substitution and poly-space. In: Computer Science Logic (CSL) 94. pp. 486–500 (1994)

24. Leivant, D., Marion, J.Y.: Evolving graph-structures and their implicit computational complexity. In: ICALP, Part II. pp. 349–360 (2013)
25. Marion, J.Y.: A type system for complexity flow analysis. In: Logic in Computer Science (LICS) 2011. pp. 123–132. IEEE (2011)
26. Marion, J.Y., Péchoux, R.: Complexity information flow in a multi-threaded imperative language. In: Theory and Applications of Models of Computation (TAMC) 2014. pp. 124–140. LNCS, Springer (2014)
27. Moser, G., Schaper, M.: From Jinja bytecode to term rewriting: A complexity reflecting transformation. Information and Computation **261**, 116–143 (2018)
28. de Naurois, P.J.: Pointers in recursion: Exploring the tropics. In: Formal Structures for Computation and Deduction (FSCD) 2019. pp. 29:1–29:18 (2019)
29. Parr, T.: The definitive ANTLR 4 reference. Pragmatic Bookshelf (2013)
30. Parr, T., Harwell, S., Fisher, K.: Adaptive LL(*) parsing: the power of dynamic analysis. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 2014. pp. 579–598 (2014)
31. Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. ACM Trans. Program. Lang. Syst. **29**(3), 15 (2007)
32. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. Computers & Mathematics with Applications **7**(1), 67–72 (1981)
33. Shkaravska, O., Kersten, R., van Eekelen, M.C.J.D.: Test-based inference of polynomial loop-bound functions. In: Principles and Practice of Programming in Java (PPPJ) 2010. pp. 99–108 (2010)
34. Spoto, F., Mesnard, F., Payet, É.: A termination analyzer for java bytecode based on path-length. ACM Trans. Program. Lang. Syst. **32**(3), 1–70 (2010)