UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Tõnis Hendrik Hlebnikov

Evaluating CodeQL for Automated Runtime Complexity Approximation

Master's Thesis (30 ECTS)

Supervisor: Vesal Vojdani, PhD

Tartu 2023

Evaluating CodeQL for Automated Runtime Complexity Approximation

Abstract:
Runtime complexity describes the execution time of a program as a function of its input size. Manual derivation of it is complex and time-consuming. As a result, automated tools that are less complex and time-consuming to use have been developed. The objective is to evaluate whether CodeQL, a tool primarily used for security analysis, could additionally facilitate runtime complexity analysis. This is evaluated through the implementation of a minimal proof-of-concept analysis which is able to provide correct estimations for some Java programs. The implications of this tool, and the process of creating it, are discussed.

Keywords:
static analysis, time complexity, CodeQL

CERCS: P175 - Informatics, Systems theory

CodeQL-i hindamine ajalise keerukuse automaatseks hindamiseks

Lühikokkuvõte:
Ajaline keerukus kirjeldab programmi käitusaega funktsioonina tema sisendi pikkuse suhtes. Selle käsitsi hindamine on keerukas ning ajamahukas. Tulenevalt on loodud mitmeid tööriistu, mille kasutamine on vähem keerukas ning ajamahukas. Eesmärk on hinnata kas turvalisuse analüüsi tööriista CodeQL-i, saaks kasutada ajalise keerukuse hindamiseks. See saab tuvastatud läbi minimaalse tööriista loomise, mis suudab korrektselt hinnata lihtsamate Java programmide ajalist keerukust. Järgneb arutelu tööriista ning loomisprotsessi ümber.

Võtmesõnad:
staatiline analüüs, ajaline keerukus, CodeQL

CERCS: P175 - Informaatika, süsteemiteooria

Contents

1 Introduction

Runtime complexity is a desirable property to know. It can provide us with answers to why a certain algorithm is faster than another, why some passwords are safer than others, and why a government website might become unresponsive under heavy load. It is one of the core concepts taught in algorithms courses, and for good reason. It assists in understanding the need for writing performant software. However, it is also one of the most demanding topics for students, as deriving runtime complexity quickly becomes a rigorous exercise in mathematics. While it would be nice to know what the complexity of a freshly minted program might be, it is simply not feasible to calculate for every change to the source code.

To combat this infeasibility, automated tools have been developed. These tools often impose strict limitations regarding what subset of language features can be analyzed, and expanding that subset is an active field of research. Several base methodologies exist, of which the primary one of interest in this thesis is inferring runtime complexity with static program analysis. Specifically, the possibility of inferring it by utilizing a relatively new tool: CodeQL.

CodeQL is generally used for security analysis and is thus primarily tooled as such. Most of all, it provides an interesting approach to static analysis by treating source code as a queryable database, upon which complex relationships can be modeled. But can it model the relationships required for the inference of time complexity? If so, to what extent? If not, why?

That comprises the primary research objective in this thesis: to determine the feasibility of implementing such analysis with the tools provided by CodeQL. To evaluate this feasibility, a tool to estimate the time complexity of simple Java programs is implemented. The implementation is detailed in section 3. Section 4 provides a discussion about the implications of this implementation.

2 Background

This section provides a definition for runtime complexity, the central topic of this thesis. Additionally, it provides a brief overview of related work in the field and an introduction to CodeQL, the semantic analysis tool being explored.

2.1 Runtime complexity

Runtime complexity provides a method for reasoning about the performance of software. This is done by expressing the time an algorithm takes to complete, either in terms of actual time or executed instructions, as a function of the size of its input.

The common representation of this is big-*O* notation, which describes the upper bound of the growth of a function. A formal definition is provided by Michael Sipser [1] as follows:

Definition. Let f and g be functions $f, g : \mathbb{N} \to \mathbb{R}^+$. Say that $f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$, $f(n) \leq c\, g(n)$.

Knowing these bounds can inform the choice of algorithm for a particular task, or about bottlenecks in a system. These bounds become critical knowledge in real-time systems, such as embedded software for engines or medical equipment. These systems often have strict requirements for responsiveness.

2.2 Automated analysis

Given the desirability of known runtime bounds, automated inference of such bounds is an active field of research. However, as computing runtime bounds for arbitrary programs is proven to be undecidable [2], the difficulty becomes one of balancing soundness (only proving properties that are true) and completeness (covering all possible cases). There are approaches that provide provably accurate bounds on runtime yet only for a restricted set of program instructions. At the other extreme of this spectrum, there are also tools that can provide approximate bounds on most programs yet do not deliver any guarantees regarding the correctness of these bounds.

2.2.1 Methods based on dynamic analysis

A common approach is based on the directly perceivable property of time complexity, that as the input size increases, the execution takes longer. Several tools have been developed that measure the execution time either by direct invocations [3] or by instrumenting the methods with timers [4]. After accumulating sufficient measurements, the complexity class can be approximated by curve fitting the measurements. While these methods can

generally cover a broad range of language features, they can not infer precise bounds and do not generally model scenarios with multiple input parameters. Additionally, soundness cannot be guaranteed, as the fastest-growing term may reveal itself at unfeasible input sizes. Although there is an argument to be made, that, for practical purposes, we might only be interested in the complexity within feasible time ranges. Nevertheless, a more invasive method is instrumenting the program with counters, upon which more precise bounds can be inferred [5].

2.2.2 Methods based on static analysis

More precise bounds can be obtained with approaches based on static analysis. The general approach using static analysis can be outlined as interpreting program instructions as a system in a more logically restricted domain. Many such domains have been evaluated for the automated inference of various properties, including time complexity. However, these restrictions ultimately extend to the original set of instructions, and this generally presents a trade-off between what can be inferred and what parts of the original program the inference can cover. This, then, extends itself to the final implementation in the form of what guarantees of soundness and completeness it can provide. A perhaps illustrative example of this transfer of limitations is demonstrated with Alan [6], a general-purpose programming language that claims predictable runtime for all computation by disallowing unbounded recursion and iteration.

Early efforts with this approach primarily tackled functional programming languages, by directly mapping expressions into symbolic cost expressions, with input sizes as variable terms, from which difference equations can be derived and solved [7]. This foundational approach has been extended to imperative languages with more involved control-flow analysis [8], deriving cost relations with regard to potential execution paths. The domain of what can be analyzed by these methods continues to expand.

2.2.3 Methods Based on artificial intelligence

With the popularity of language models, it is anticipatable that they would be utilized for time complexity estimation [9]. Understandably, any approach based on language models is unlikely to make any claims of soundness. However, if the requirement for precision is sufficiently loose, there is an argument to be made in favor of having an approach that is not restricted in its input.

2.3 CodeQL

The CodeQL website[1] describes CodeQL as a tool for the semantic analysis of codebases. It is designed to facilitate querying source code in a similar manner to how one would query common relational databases - as if source code was data. It was originally developed by the company Semmle [10], which has since, along with the software, been acquired by GitHub [11]. GitHub has subsequently integrated CodeQL scanning on the platform as a feature available to all public repositories and private ones with the appropriate license. It is additionally made publicly available, as stand-alone software, with a license that authorizes use for, among other things, academic research[2].

The analysis itself is expressed as queries. Queries are constructed such that they describe a notable property of the source code by modeling the property through relationships on an abstraction of said source code. These abstractions enable decoupling the queries from individual codebases and enable their reuse. Thus, a query constructed with the intent of detecting a noteworthy property in one codebase may be used to detect the same property in another codebase. Collections of such universal queries are published as packs, either as query packs to execute directly or library packs to build upon. Published queries additionally contain metadata to inform their interpretation, such as their estimated precision and assessed severity.

CodeQL is commonly used to automate the detection of security issues. The bundled language-specific libraries contain numerous features which, along with the expressive language itself, make it convenient to describe many common vulnerabilities. For example, the default query pack provided for Java[3] contains queries to detect potential vulnerabilities to SQL injection, cross-site scripting, and session hijacking. An overview of Mitre's Common Weakness Enumeration (CWE) coverage by CodeQL is provided on the website[4]. In addition to security issues, the aforementioned pack contains queries to detect various other issues. Some examples of these, that demonstrate the range covered by CodeQL, are queries to detect unreachable code, inefficient code, such as string concatenation in a loop, and violations of best practices, such as commented-out code. As such, its usefulness extends beyond security analysis. It is ultimately designed as a tool to inform and streamline manual code review.

Another noteworthy advantage of using CodeQL for such analysis is its large com-

[1] CodeQL Website: `https://codeql.github.com/`

[2] CodeQL CLI license: `https://github.com/github/codeql-cli-binaries/blob/main/LICENSE.md`

[3] Query pack available as source code on GitHub: `https://github.com/github/codeql/tree/main/java/ql/src`

[4] CodeQL CWE coverage: `https://codeql.github.com/codeql-query-help/full-cwe/`

munity. With the integration by GitHub, CodeQL is made available to a large audience. While some of the source code is unavailable, a large part of it is open source. This enables collaborative development with potential contributors who are not directly associated with GitHub. As new vulnerabilities are discovered, the CodeQL community can iterate upon previous work to cover these vulnerabilities. This is further promoted by GitHub's bounty programs[5]. Additionally, as these contributors often document and publish their experiences, the barrier of entry is lowered.

2.4 QL

The query language for CodeQL is QL. It is a declarative, object-oriented logic programming language. At first glance, it holds many notable similarities to SQL, which is a design choice to lower the barrier of entry for potential developers. However, these similarities do not extend beyond syntax, as the semantics are based on Datalog - a declarative logic programming language. Furthermore, as QL programs, the queries, are run in CodeQL, they are first compiled into a variant of Datalog. A thorough overview of the language may be obtained from the language reference provided by GitHub[6] and the original introductory papers [10, 12, 13]. The following sections provide a brief overview of language features relevant to the thesis.

2.4.1 Predicates

A program in QL can be viewed as the declaration of a set of relations. These relations are described through the use of predicates. Explicit predicates in QL come in two forms: without, and with a result. Predicates without a result are used to model the properties of its terms. Predicates with a result, reminiscent of functions in other programming languages, define a result variable that is returned upon their evaluation. Further relations may then be expressed regarding the result. An example of a predicate in QL is provided in listing 1.

```
bindingset[n]
predicate isPrime(int n){
    n > 1 and
    not exists(int divisor
        | divisor in [2 .. n - 1]
        | n % divisor = 0
    )
}
```

Listing 1. Predicate to determine primality.

[5]CodeQL Bug Bounty program: https://securitylab.github.com/bounties/
[6]QL language reference: https://codeql.github.com/docs/ql-language-reference/

The result set of a query is required to be finite, and this consequently applies to the predicates. This requirement can be satisfied by binding its variables to finite ranges. Alternatively, it may be delegated to the usage of the predicate by annotating it with `bindingset`. The annotation is needed for `isPrime` as the set of inputs that it holds for is infinite. The predicates themselves may be recursive, or they may be applied recursively.

2.4.2 Object-orientation

Classes in QL represent a logical property. The common approach in object-oriented languages is that objects are instantiated by allocating physical memory to maintain object state. This is not the case for QL, however, as classes are representations of a logical property. Thus, instead of providing a template for new instances, classes describe subsets of existing values. Instead of constructors, classes in QL define characteristic predicates. These are syntactically similar to constructors, yet define members of the class by declaring logical restrictions on the `this` variable. For example, the predicate `isPrime` from earlier may be adapted into the characteristic predicate for a class, as shown in listing 2.

```
class Prime extends int{
    bindingset[this]
    Prime(){
        this > 1 and
        not exists(int divisor
            | divisor in [2 .. this - 1]
            | this % divisor = 0
        )
    }
}
```

Listing 2. Class to represent prime numbers.

Classes must always extend a supertype. If the supertype defines a characteristic predicate, it must also hold. Inheritance is thus modeled as a logical implication. In the case of multiple inheritance, the set of values which may belong to a class is the intersection between the sets of values of its superclasses. Classes may define and override member predicates and fields. These are resolved at run time by picking the most specific class definition. If there are multiple such definitions, the predicate is evaluated for all of them and a union of the result sets is returned. A class may additionally be declared as abstract. Then, a member value must additionally satisfy the characteristic predicate of at least one of its subclasses.

2.4.3 Queries

Ultimately, collections of predicates describe an interesting logical property. To obtain a set of actual values with this property, a query must be constructed. Queries come in the form of query predicates, or more commonly select clauses. Select clauses declare variables, establish restrictions on them as logical formulas which must be satisfied, and evaluate them to concrete values. An example query is provided in listing 3.

```
from Prime n
where n in [1 .. 100]
select n
```

Listing 3. QL query that finds all primes in the range [1, 100.]

The example utilizes the `Prime` class from earlier and restricts the possible values to the range [1, 100]. Any number of variables may be declared and the formulas may be arbitrarily complex.

2.5 Static analysis with CodeQL

The declarative nature of its query language extends to the static analysis performed with CodeQL. The program properties to be discovered are provided to the query execution engine as instructions of *what* should be computed, rather than *how*. These instructions are then evaluated against a database that contains a relational representation of the source code. The process is evocative of querying common relational databases and pattern matching on graph databases. A simple example for Java source code, using the `Prime` class from earlier, is provided in listing 4.

```
import java
import semmle.code.java.dataflow.RangeUtils

from DivExpr expr, IntegralType t
where
    expr.getLeftOperand().(ConstantIntegerExpr)
        .getIntValue() instanceof Prime and
    expr.getType() = t
select expr
```

Listing 4. Query that selects all expressions where an integer, which can be evaluated statically to be both constant and a prime, is divided, and the result is interpreted as an integer.

Concrete elements of the source code are represented by abstractions provided by the language-specific CodeQL library. Libraries are provided for many popular languages,

and in some cases even for specific frameworks for these languages. The abstractions in these libraries are generally similar, yet ultimately incompatible, due to the differences between the source languages. This extends to the analysis itself, meaning that different source languages may require different approaches to express the same property. A methodology has been proposed to derive unified cross-language abstractions [14].

The queryable representation of a codebase is produced by an extractor. For compiled languages, extractors are attached to the compiler and obtain relevant information by monitoring the build process. Source code is parsed directly for interpreted languages. The extractor additionally resolves dependencies and stores the obtained relations in a database. The database is referred to as a *snapshot database* since it contains information about a program at a single point in time. Databases are created by GitHub for many popular open-source projects and may be downloaded. Alternatively, they may be created by running the extractor locally.

The specific schema for these databases largely depends on the source language. Documentation provided by GitHub states that a public-facing specification is unavailable. However, they are stated to contain:

- a complete representation of the source code,
- the abstract syntax tree,
- the data flow graph,
- the control flow graph,
- query results,
- log files.

Most of this content is encoded in a binary format and is intended to be observed through the execution of queries. The exception to this is the abstract syntax tree, which may be explored in Visual Studio Code[7] with the CodeQL extension[8].

The CodeQL standard library provides many common analyses out-of-the-box. The following sections provide an overview of the ones relevant to the thesis. The overview is based on the documentation and source code for the library provided for Java. Thus, the discussed specifics may not hold for the libraries for other languages.

[7]Visual Studio Code: `https://code.visualstudio.com/`

[8]CodeQL extension: `https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-codeql`

2.5.1 Control-flow analysis

Control flow analysis is the process of determining possible paths through program instructions. Possible in the sense, that certain constructs may split subsequent flow based on a condition. For an accurate representation, these conditions are evaluated within the context of previously satisfied conditions. It is often useful to interpret this as a graph, where nodes represent program statements and edges represent possible flow.

CodeQL provides this analysis primarily through the use of the class `ControlFlowNode`. Possible flow is modeled through the member predicate `getASuccessor`. This predicate may be applied recursively to determine whether there is a path from one node to another. A subclass of this is `ConditionNode`, which denotes the expression at which the subsequent flow splits. It provides predicates to obtain the successor in a specific branch, as well as the condition expression itself. It is worthy of note, that the analysis is path-sensitive. When the predicates are used to model a path, CodeQL can infer constraints that must hold for the path to exist. However, the documentation does not provide information regarding what kinds of constraints may be evaluated, and the classes themselves do not provide access to these constraints either.

It is additionally of note, that the predicate only returns successors that reside within the same method invocation; or that the analysis provided is intraprocedural.

2.5.2 Static single-assignment form

Static single-assignment (SSA) form is the representation of a program such that variables are assigned only once. Converting a program to such a representation is accomplished by replacing all variable updates with declarations and reads of these variables with reads of their most recent declaration. If there are multiple most recent declarations to choose from, the appropriate choice is informed by prior control flow.

CodeQL makes extensive use of SSA internally and also exports various classes for it in the `SSA` module. The implementation discussed in section does not make significant direct use of the logic provided, yet it does use some of the declared classes. These are then described within the context of their usage.

2.5.3 Value range analysis

Value range analysis is the process of determining the range of values which a numeric variable may hold at certain points of program execution. The bounds of this interval are generally symbolic expressions in terms of other variables.

CodeQL provides bound inference with the `RangeAnalysis` module. The module contains the predicate `bounded(Expr e, Bound b, int delta, boolean upper, Reason reason)`. This predicate can be expressed as:

$$bounded(e, b, \delta, upper) \iff \begin{cases} e \leq b + \delta & \text{if } upper \\ e \geq b + \delta & \text{if} \neg upper \end{cases}$$

The documentation states that the inferred bound may be a specific integer, the abstract value of an SSA variable, or the abstract value of an interesting expression. The last of these requires further explanation. The `Bound` class has three direct subtypes, respectively: `ZeroBound`, used to represent the specific integers, `SsaBound`, and `ExprBound`. The documentation for `ExprBound` states that it "corresponds to the value of a specific expression that might be interesting, but isn't otherwise represented by the value of an SSA variable". Inspection of the source code reveals that this may be the length of an array.

The source code additionally reveals that these bounds can be inferred across steps of additions, subtractions, strict comparisons, and loop iterations. These steps, if possible, are then summed as the integer delta. A non-constant literal bound itself represents a single SSA variable.

The implementation discussed in section 3 makes significant use of this module to infer loop iterations.

3 Design and implementation

CodeQL provides an extensive library of static analysis tools. The development of this library has primarily been influenced by requirements for security analysis. The primary question, then, is whether it is feasible to utilize this library to analyze runtime complexity. This feasibility shall be evaluated through the implementation of a tool that serves as a proof of concept. The broad requirement for such a proof-of-concept tool is that there must exist a program, consisting of a non-empty subset of Java language features, for which it can provide the correct runtime complexity. With consideration of the features available, the proof-of-concept must:

1. detect loops;
2. determine an upper bound on loop iterations, or report that it is unable to do so;
3. provide a meaningful summary of these bounds in terms of an input parameter.

The feasibility is then considered as the difficulty of satisfying these requirements without significant extension to the standard library. This is based on the notion that more complex analysis has been implemented in more appropriately specialized tools.

3.1 Implementation

This section describes how the relevant features of CodeQL are utilized and provides details of the implemented tool. Broadly, the tool attempts to infer bounds for loop control variables and the nesting relationship of these loops. If it can do so, it provides a representation of these bounds and the nesting. If it is unable to do so, it reports as such. It is comprised of four classes, for which a class diagram is provided in figure **??**, and a non-member predicate.

The specifics of the predicate and classes are described in the following subsections. The capabilities of the implemented tool are discussed, along with examples, in section 3.2. The QL source code for the implemented tool is provided in the accompanying files.

3.1.1 Loops

CodeQL provides a class for explicit loop statements as `LoopStmt`. In order to encapsulate the logic for their analysis and to facilitate interpretation later, a class `Loop` is created which extends `LoopStmt`. This inherits two predicates from `LoopStmt` that are immediately relevant: `getBody`, which returns the statement for the loop body, and `getCondition`, which returns the expression that serves as the condition for this loop.

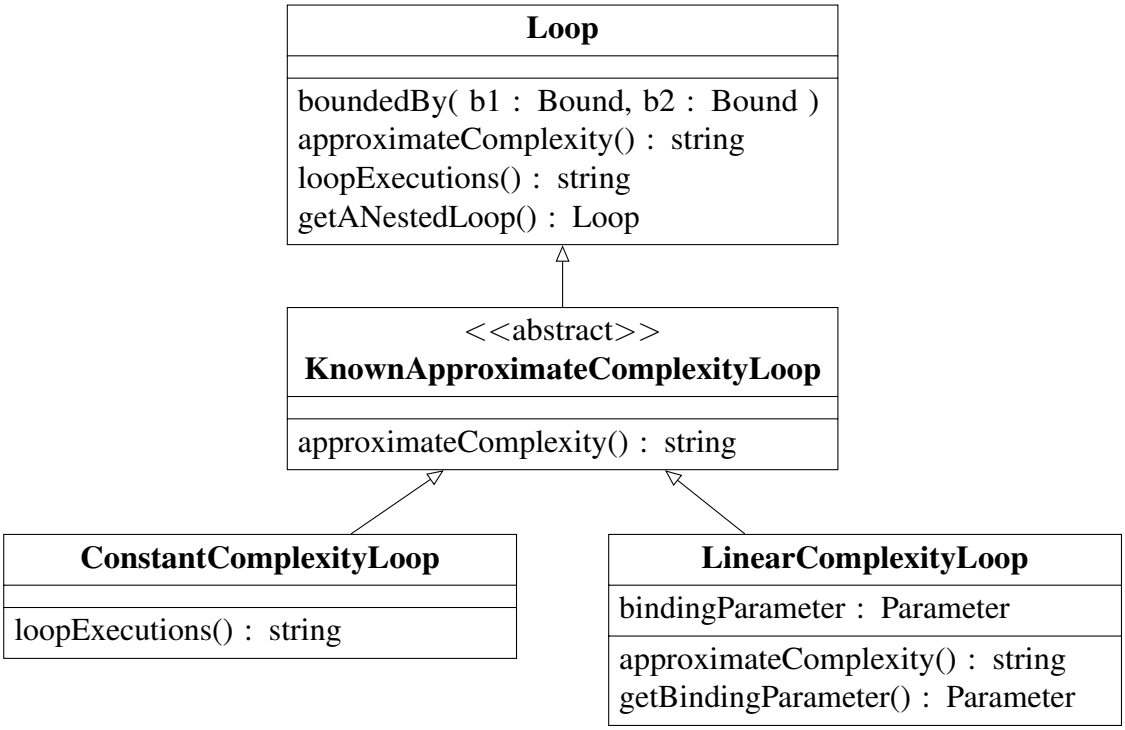

Figure 1. Class diagram of the implementation.

In order to determine the bounds of the iteration, the member predicate `boundedBy` is defined as:

$$\begin{aligned} boundedBy(b_1, b_2) \iff \exists c, r, x, i, u, s : r = getAFirstUse(c) \wedge \\ x = getCondition() \wedge \\ r = getAChildExpr(x) \wedge \\ u = getDefiningExpr(i) \wedge \\ backEdge(c, i) \wedge \\ bounded(u, b_1, s) \wedge \\ bounded(u, b_2, \neg s) \end{aligned}$$

where $b_1, b_2 \in Bound, c \in SsaVariable, r \in RValue, x \in Expr,$
$i \in SsaExplicitUpdate, u \in VariableUpdate, s \in \{true, false\}.$

`getAFirstUse` is a member predicate of the class `SsaVariable` that returns an access of the same SSA source variable. `getAChildExpr` is a member predicate of `Expr` and returns a child of the expression. `SsaExplicitUpdate` is defined as an SSA variable

that is created by a variable update. Its member predicate `getDefiningExpr` returns the corresponding expression as a `VariableUpdate`. The predicate `backEdge` holds for c and i if i is an input to c along a back edge. This is used here to detect the point at which a control variable, the variable that determines loop iterations, is updated. The `bounded` predicate is then used to determine whether the variable update expression is bounded, such that $b_1 \leq u \leq b_2$. The usage of the $boundedBy$ predicate will be clarified in more detail in the following sections.

The class additionally declares the member predicates `approximateComplexity` and `loopExecutions`, which are used for summarizing the iterations. They are intended to be overridden when such a summary is available, and for `Loop` simply return the string `"?"`. The member predicate `getANestedLoop` returns for a `Loop` l_1 another `Loop` l_2, such that l_2 is declared within the body of l_1.

The subclass `KnownApproximateComplexityLoop` is created to represent loops for which iteration bounds can be determined. The class is declared as abstract. It exists to be further extended by more concrete subclasses. The characteristic predicate holds if all of the loops nested within are subclasses of `KnownApproximateComplexityLoop`. The member predicate `approximateComplexity` overrides the one from `Loop` and returns a string summary of the approximate complexity. It is defined such that if there are no nested loops, it returns the bounds for itself by calling the `loopExecutions` member predicate of its superclass. If there are nested loops, it returns the concatenation of its own summary with the summary of nested loops.

3.1.2 Linear loops

The class `LinearComplexityLoop` represents loops that are estimated to have a linear bound on their iterations regarding a `Parameter`. This `Parameter` is stored in the class field `bindingParameter`. The overriding member predicate `loopExecutions` returns the variable name of this `Parameter`, as it is in the source code.

The binding parameter is determined within the characteristic predicate of the class. The predicate holds if there exists an `SsaBound` b and a `ZeroBound` z, such that b is a parameter and $boundedBy(b, z)$ holds. The `ZeroBound` z represents a constant value. The interpretation is that if the `RangeAnalysis` module determines that the control variable is bounded at one end by z and at the other end by b, as they are both integers, there are at most $|b - z|$ values it may hold. Since z is a constant, the only variable term is b. Thus, b is interpreted as being the binding, and thus summarizing bound of the given loop.

3.1.3 Constant loops

The class `ConstantComplexityLoop` represents loops that are estimated to have constant bounds. It overrides the member predicate `loopExecutions` to report as such. The characteristic predicate of the class holds if there exists a bound b, such that b is either a `ZeroBound` or an `SsaBound` representing a parameter, for which $boundedBy(b, b)$ holds. If b is a constant, the range of values is constant. When b represents a parameter, the count of possible values may be expressed as $|(b + \delta_1) - (b + \delta_2)|$, where δ_1 and δ_2 are integer deltas determined by the library. As the deltas must be constant, the count of iterations is constant.

3.1.4 Interpretation

As we now have bounds for loops, it follows to provide a meaningful representation of them. A representation for a particular loop, including the ones it nests, is obtained through the `approximateComplexity` member predicate. A similar representation may be expressed for methods by considering the loops they contain. This is implemented as the non-member predicate `getApproximateComplexity(Method m)`. If the provided method does not contain any loops, then its runtime complexity is interpreted to be constant. If it contains a loop for which an estimation is unavailable, it reports the complexity as unknown. If all of the loops it contains have estimated bounds, it concatenates the representations for all top-level loops. An example of the results obtained from this predicate is provided in figure 2. The methods which it can not provide an estimation for are discussed in the next section.

#	function	summary ▼
1	On_arr_1	Ö(x)
2	On_arr_2	Ö(x)
3	arrarg	Ö(x)
4	onloop	Ö(x)
5	On_1	Ö(x)
6	problematic24	Ö(x(y))
7	problematic8	Ö(x(x+x))
8	problematic14	Ö(x(x+x(1)))
9	On2_2	Ö(x(x))

Figure 2. A result obtained while testing `getApproximateComplexity`.

It is additionally noticeable from the example that these approximations are not reported as big-O, but instead, as big-$\ddot{O}$, as they do not directly conform to the common notation. This representation was chosen in order to show the loops and their nesting relationships explicitly. The symbol was chosen as $\ddot{O}$in reference to both big-$\tilde{O}$ and the vocalization for a common hesitation marker.

3.2 Evaluation

Throughout the development of the proof-of-concept tool, 63 simple Java functions were written. These served as the basis for the iteration toward a working implementation. The final implementation can provide a meaningful representation for 34 of them, which if interpreted as runtime complexity, is correct. This includes implementations of simple sorting algorithms. It reports a strictly wrong summary for 7. For the rest it reports that it is unable to provide an estimation. This section provides examples and a discussion of the more notable cases among these.

The function for which it reports the wrong bound is shown in listing 5. This is due to the $O(n)$ method call within the loop. The tool decidedly does not consider method calls, as the standard library does not provide sufficient features for this. A refinement to cover these cases was made at a point in development in the form of a check, that all method calls within a loop must have compile-time constant arguments. This refinement was ultimately commented out, however, as the implementation is not meant to cover method calls. A similar case with a different implication is shown in listing 6. The function does not contain a loop statement and is thus falsely reported as having constant complexity.

```
public static int p17(int x){
    for (int i = 0; i < x;i++) {
        onloop(x);
    }
    return 0;
}
```

Listing 5. A function falsely reported as having linear complexity.

```
public static int Onrec1(int x){
    if (x < 5)
        return x + 3;
    return Onrec1(x-1);
}
```

Listing 6. A function falsely reported as having constant complexity.

The cases where it reports as unable to provide an estimation are primarily ones where the bound is a value that the `RangeAnalysis` module is unable to represent, such as the result of a multiplication. Listing 7 provides an example of such a function. Another issue presents itself when the control value is updated by an expression containing a function call. An example of this is provided in listing 8. The module does not determine bounds for the control variable as it does not support interprocedural analysis.

```java
public static int On_2(int x){
    int y = 2 * x;
    for (int i = 0; i < y; i++) {
        System.out.println("ok");
    }
    return 0;
}
```

Listing 7. A function reported as having unknown complexity.

```java
public static int add1(int z){
    return z+1;
}
public static int p15(int x){
    for (int i = 0; i < x;) {
        i = add1(i);
    }
    return 0;
}
```

Listing 8. A function (p15) reported as having unknown complexity.

A notable example of where it is correct is a version of bubble sort, shown in listing 9. It is reported as being $\ddot{O}(a(a))$. This shows that it can interpret a bound even if the parameter is not an integer, but has a field that is. Another example is shown in listing 10, where the parameter x is used to specify the length of an array. This function is reported as being $\ddot{O}(x(x))$. However, this does not extend to multidimensional arrays.

```java
public static void bubbleSort(int[] a){
    for (int i = 0; i < a.length; i++) {
        for (int j = 1; j < a.length; j++) {
            if (a[j-1] > a[j]) {
                int temp = a[j - 1];
                a[j - 1] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

Listing 9. A version of bubble sort.

```java
public static int On2_arr_2(int x){
    int[] a = new int[x];
    for (int i = 0; i < a.length; i++) {
        for (int j = 0; j < a.length; j++) {
            System.out.println(i == j);
        }
    }
    return 0;
}
```

Listing 10. A function where the parameter is used to specify the length of an array.

In some cases, it can establish a bound for the control variable if the loop has multiple exit conditions. An example of this is a version of insertion sort, shown in listing 11.

```
public static void insertionSort(int[] a){
    for (int i = 1; i < a.length; i++) {
        for (int j = i; j > 0; j--) {
            if (a[j-1] > a[j]) {
                int temp = a[j - 1];
                a[j - 1] = a[j];
                a[j] = temp;
            } else break;
        }
    }
}
```

Listing 11. A version of insertion sort.

Multiple parameters are also supported, provided that they do not interact directly. Listing 12 depicts a function for which $\ddot{O}(x(y))$ is reported.

```
public static int problematic24(int x, int y){
    for (int i = 0; i < x; i++){
        for (int j = 0; j < y; j++) {
            System.out.println("ok");
        }
    }
    return 0;
}
```

Listing 12. A function with multiple parameters.

The code depicted in these examples is exactly as it was written during development, with the only changes being removal of some whitespace, and shortening the name of one. The source code for all of the functions used for testing the implementation, whether depicted in the examples or not, is provided in the accompanying files.

These examples serve to illustrate that the implemented tool satisfies the requirements set for a proof of concept; that it can correctly determine the runtime complexity for a program composed of a subset of language features. As it makes a decision to overlook several features which may affect runtime complexity, it is not intended to be used in any real-world scenarios. However, it is likely to provide the correct estimation for many functions in a Java codebase. The implication of this is discussed in section 4.

4 Discussion

The final implementation described in section 3 is the result of significant trial and error. It serves as a proof of concept, and the process provides some indication of whether runtime complexity analysis with CodeQL is feasible. To the author's knowledge, this was the first attempt to perform such an analysis with CodeQL. This extends to say that the relevance of available features was not initially known.

The design of the QL language has reportedly been influenced by the desire to lower the barrier of entry for new developers. While it is reminiscent of many common query languages, it is fundamentally different. It presents a steep learning curve for anyone not formerly acquainted with logic languages. Expressing an interesting property in QL is often a non-trivial endeavor.

The difficulty is substantially supplemented by the standard library of CodeQL. It provides very little in the way of documentation for most of its features. Many features are accompanied by a single line of documentation, and some are deprived of even that. It is often necessary to inspect the source code to obtain an overview of a feature. For a comprehensive understanding, this becomes critical. However, the functionality of a feature is often divided up as a deep class hierarchy, which must then be navigated.

The design of the implementation was thus iterated upon through exploration of what was possible with the library. Knowing that CodeQL provides path-sensitive control flow analysis, the first idea was to observe the constraints on these paths and determine whether they contain enough information to make assumptions about runtime complexity. Through experimentation, it became apparent that these constraints are not exposed directly. Further, if these constraints could be exposed, there is no built-in way to represent or evaluate systems of symbolic expressions. It became apparent that any approach within scope would have to rely on the `RangeAnalysis` module.

The control flow analysis provided is not interprocedural. However, the data flow analysis is. Thus, an attempt was made to use it to track interprocedural flow. This proved troublesome as it is less precise, and in most cases of interest, the value is not preserved and thus no longer tracked. The attempt was ultimately abandoned, as the range analysis is not interprocedural either.

With that said, however, there is a case to be made for the advantages of CodeQL. The language itself is very expressive and complex properties can be described in a concise manner. While the standard library is difficult to navigate, the source code itself is rather legible. Furthermore, as CodeQL analysis itself is integrated into the GitHub platform, any analysis implemented could benefit a large audience. Executing analysis

with the click of a button on GitHub is decidedly simpler and more convenient than setting up a dedicated tool.

CodeQL has a very active community that contributes to its development. Issues on its GitHub repository are opened and resolved daily, and a search for the term "CodeQL" on Google Scholar[9] yields 99 results since 2023. It is constantly evolving, and it is likely that features necessary for more complex runtime complexity inference will eventually be implemented. It is not unreasonable to predict that the standard library could eventually even include a form of such analysis. Then, as open source, it could be iterated upon by the community. The analysis could additionally be completely automated by the platform to run on every commit. Perhaps it would even provide a badge to display in the `README.md` file of a repository.

[9]Google scholar: `https://scholar.google.com/`

5 Conclusion

CodeQL is a static analysis tool primarily used for the detection of security issues. It comes equipped with an extensive standard library for that purpose. It was postulated, that perhaps this standard library could be used for runtime complexity analysis. This formed the primary research objective; to ascertain whether such analysis could be done, and to what extent. This would be evaluated through a proof of concept implementation.

To inform the design of such an implementation, prior work in the field was reviewed alongside the documentation and source code of CodeQL itself. An approach was established and iterated upon. Throughout experimentation, it became apparent that the library does not provide adequate tooling out-of-the-box, and that analysis on par with specialized tools would not be feasible to implement. Despite this, a tool that could provide meaningful approximations for some functions could be devised.

Such a tool was implemented and its capabilities and limitations were discussed. The implications of the tool and its limitations, as well as the process of creating it, serve to form an answer to the research question. Ultimately, the standard library would require extension to constitute the basis of a more accurate tool.

References

[1] Michael Sipser. *Introduction to the theory of computation*. Course Technology, Florence, AL, 3 edition, 2021.

[2] Andrea Asperti. The intensional content of rice's theorem. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2008. ACM.

[3] Pietro Berkes. big_o: Python module to estimate big-o time complexity from execution time.

[4] Istvan Czibula, Zsuzsanna Oneţ-Marian, and Robert-Francisc Vida. Automatic algorithmic complexity determination using dynamic program analysis. In *Proceedings of the 14th International Conference on Software Technologies*. SCITEPRESS - Science and Technology Publications, 2019.

[5] Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. *SIGPLAN Not.*, 44(1):127–139, 2009.

[6] Alan language - alan language. `https://alan-lang.org/`. Accessed: 2023-8-10.

[7] Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[8] Reinhard Wilhelm. Determining bounds on execution times. In *Embedded Systems Handbook*, 2005.

[9] Kaushik Moudgalya, Ankit Ramakrishnan, Vamsikrishna Chemudupati, and Xing Han Lu. TASTY: A transformer based approach to space and time complexity. 2023.

[10] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address: .ql for source code analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. IEEE, 2007.

[11] Nat Friedman. Welcoming semmle to github, Sep 2019.

[12] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented queries on relational data. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2016.

[13] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. .ql: Object-oriented queries made easy. In *Generative and Transformational Techniques in Software Engineering*, 2007.

[14] Dongjun Youn, Sungho Lee, and Sukyoung Ryu. Declarative static analysis for multilingual programs using codeql. *Software: Practice and Experience*, 53(7):1472–1495, 2023.

Appendix

I. Accompanying files

1. The QL **source code** is "**BigÖ.ql**", found in the archive "BigÖ.zip" .
2. The **java functions** used to test the implementation are in "**Tests.java**", found in the archive "BigÖ.zip" .

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Tõnis Hendrik Hlebnikov**,
*(*author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Evaluating CodeQL for Automated Runtime Complexity Approximation,
   *(*title of thesis)

   supervised by Vesal Vojdani.
   *(*supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Tõnis Hendrik Hlebnikov
11/08/2023