Quantifying the Benefits of Object-Oriented Languages with Higher Order Domains

Nick Cooper , Neelima Prasad , Srikrishna B.R.

University of Colorado at Boulder
nico5120@colorado.edu, nepr1244@colorado.edu, srba2850@colorado.edu

Abstract

The structure of software is an important aspect of real world systems that affects factors such as ease of maintenance, ease of modification, and stability. Despite the abundance of motivation, formally quantifying this notion is non-trivial; and a wide variety of methods dating back to the later half of the twentieth century have been proposed. However, many proposed analytic frameworks are not generalizable to all languages and require access to the entire code-base, which limits their applicability to real-world code analysis tasks. As programming languages differ greatly in terms of the structural tools they provide, a generalizable analytic framework is required to provide insights on their relative benefits and trade-offs. Furthermore, obtaining full code-bases is not always possible, practical, or even relevant to many use cases. To address these limitations, we introduce a novel **language-agnostic** framework for analyzing the structure of software when **only one source file** is provided. We employ combinatorial complexes, a recently introduced higher order domain, to accomplish this goal. Our technique is not only able to draw the same conclusions as existing full code-base methods, but also provides novel insights about the benefits of Object-Oriented (OO) languages.

Introduction

Analyzing the structure of software is an important task which facilitates code analysis and provides insights about the benefits and trade-offs of different languages. Formalizing this notion has been the subject of extensive research throughout the past 50 years, with a multitude of techniques proposed. A common theme is the **topology** of the software, which is commonly discussed in the language of graph theory.

Object-Oriented programming languages are powerful and ubiquitous in industry, and have thusly had immense impact on structural analysis techniques. However, this impact has resulted in many state-of-the-art methods over-specifying to narrow sub-classes of languages. This is problematic because while OO languages often exhibit similarities, they can differ drastically and each present distinct strengths and weaknesses. Furthermore, most OO-inspired frameworks cannot provide any insight as to why OO languages are preferable to functional-oriented options. To make matters worse, these frameworks typically assume access to the entire code-base. This is a serious limitation for three key reasons: (1) it can be computationally expensive to run sophisticated algorithms on the entirety of enterprise-scale code, (2) the full code-base may not be available, or providing it may induce privacy concerns, and (3), there are many useful intra-file applications of structural analysis such as evaluating code generated by LLMs.

In this paper, we introduce a novel analytic framework which addresses all the above issues. Specifically, we extend ideas from graph-based methods to a higher order domain to enable meaningful results to be gleaned from a single source file. Combinatorial complexes are our domain of choice for their ability to model a wide range of intricate structural relationships. To enable cross-language analysis, we "objectify" functions which then form the basis (minimal rank cells) of the complexes. Our method is therefore intrinsically applicable to **any** piece of code, and can be easily extended to the full code-base paradigm as well. Additionally, it is computationally cheap; and capable of processing upwards of 2000 source files per minute per logical processor (thread) [1].

We validate our technique by comparison against a standard existing framework, and observe that our method is able to provide the same conclusions. Because our framework also extracts richer information than the baseline, we then leverage its cross-language abilities to answer a series of questions about why OO languages reign supreme over their functional counterparts. Concretely, we compare Rust to two of the most popular OO languages: C++ and Java. We observe that the OO languages lead to structurally superior code which exhibits favorable characteristics such as improved coupling. We formally quantify these benefits and discuss their implications for the real world.

Related Work

Code Structure of C++ and Java

Existing work has proposed methods for quantifying the differences between Java and C++. Kumari et. al (Kumari and Bhasin 2011) highlight how languages features can influence code structure. One example is that multiple inheritance is implemented using classes in C++, while in Java,

[1] As tested on an AMD Ryzen 7800X3D CPU.

it is implemented using interfaces. Furthermore, Java supports only method overloading and doesn't allow operator overloading, but C++ supports both. As a part of dynamic polymorphism, in C++, the virtual keyword is used with a function to indicate the function that can be overridden in the derived class. This way we can achieve polymorphism. In Java, the virtual keyword is absent. However, in Java, all non-static methods by default can be overridden. It is interesting to consider how these differences influence the properties of real-world code.

To provide a baseline understanding of the prominent structural characteristics of Java and C++, we leverage seven different metrics from the framework of (Kumari and Bhasin 2011). These are: size, complexity, information hiding, coupling, cohesion, inheritance, and polymorphism. This enables us to produce a baseline analysis of Java vs. C++ which we use to validate the effectiveness of our proposed method.

Defining Topologies on Software

Modeling software with topological mathematical objects dates back to at least the 1980s (Ejiiogu 1985). Graphs have since continued to remain a prevalent choice for capturing and studying the relationships between "objects" (Pan and Chai 2017; Yutao Ma and Du 2005; Xinxin Xu and Wang 2022). Specifically, graph based models of software have been used to define notions of software stability (Pan and Chai 2017), and structural entropy (Yutao Ma and Du 2005). These ideas allow for straightforward definitions of coupling strength, abstraction depth, and other useful analytic measures.

However, state-of-the-art approaches such as (Xinxin Xu and Wang 2022) typically cannot generalize uniformly to all OO languages, let alone functional or procedural languages. This is because different languages provides distinct rules which influence the possible structures of code. As an example, because C++ does not require all code to reside inside a class, there may exist relevant structural information which class-centric frameworks like (Xinxin Xu and Wang 2022) cannot model. This limitation immediately disables the possibility of employing such tools for practical real-world interests such as informing language selection for new projects. Additionally, this style of framework requires access to the entire code-base, which creates several more limitations preventing even more use-cases. We use the recently introduced notion of combinatorial complexes (Mustafa Hajij 2023) to construct a new widely applicable analytic framework.

Methods

Setting We consider a constrained version of the typical full code-base setting of software complexity analysis. To ensure widespread applicability, we design metrics which can be computed from only the structure which is visible from one source file. See Figure 1 for an illustration. Should multiple files be available, one may simply concatenate all source files then apply our method for increased analytic power.

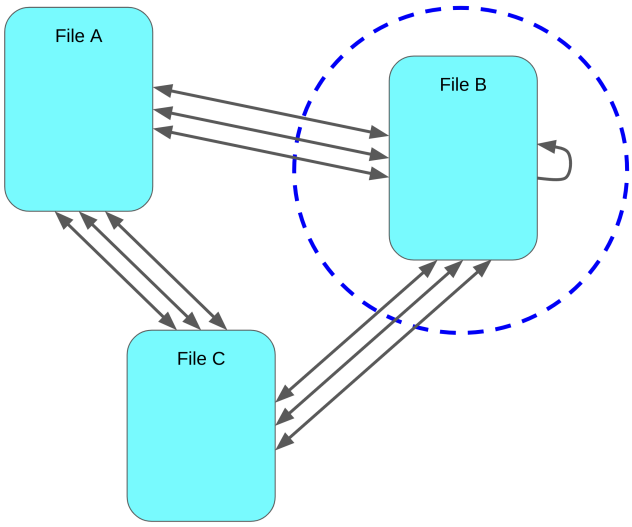

Figure 1: Visualization of the single source file constraint. Arrows indicate relationships between parts of the code; e.g. function call relations. Our method is designed to operate on one source file at a time (dashed blue circle). This enables it to generalize to a broad class of applications.

Baseline Analysis of Java vs C++ Several fundamental OO principals are described in (Kumari and Bhasin 2011) and (Awang Abu Bakar 2016). These are: size, complexity, encapsulation, coupling, cohesion, inheritance, and polymorphism. As there are many metrics that can be used to measure each of these attributes, we will focus on one metric per attribute, and describe the performances of Java and C++ in the results section measured in these papers.

For size, we will use the number of operations added by a subclass (NOA). NOA is a size metric that defines size of a class in terms of number of attributes defined in the class. As the size of an object is sum of memory consumed by all the attributes of class, then a larger number of attributes means an increase in the object size.

Complexity can be measured by computing the sum of complexity of all methods of class, or weighted methods per class (WMC). The complexity for each method is calculated traditional Cyclomatic Complexity metric. Higher values of WMC are not desirable because complex methods are more difficult to maintain and test.

Attribute hiding factor (AHF) is an encapsulation metric that counts the average amount of attribute hiding among all classes in the system. A higher value is desirable as it indicates better access management.

Coupling is measured by coupling factor (CF), which is defined as the number of actual couplings over the maximum possible couplings. A lower CF is desirable.

Lack of Cohesion in Methods (LCC) measures the percentage of pairs of public methods to the class that use common attributes (directly or indirectly). A high value of LCC is desirable as it means cohesion is high which indicates that classes are subdivided properly.

To measure inheritance, we use the Depth of Inheritance

Tree, which is the max path length from root to node in inheritance tree. A high value of DIT means more methods can be inherited, but makes it more complex.

Finally, to measure polymorphism, the polymorphism factor (PF) is calculated by dividing the number of existing method overrides by the number of possible overrides. The higher the factor, the more difficult it is to understand.

We employ these measures to provide a baseline analysis of the structural differences between Java and C++. This serves as a grounding frame of reference to confirm that our techniques provide useful information.

Proposed Method

In order to measure structural attributes of large quantities of real world software, we make "objects" less complex and demote their definition to "function". The popular class-level definition provides many interesting relations between sections of code, but there are two key limitations preventing their use in a truly generalizable framework. First, not all OO languages provide the same relations between classes. Second, there is no guarantee that developers will use the provided relations in the same way, or even in an optimal way. Therefore, we elect to make "objects" simpler which in turn permits the modeling of a wide range of higher order relations between them. We employ combinatorial complexes for this purpose, and discuss the theoretical setting next.

Theoretical Framework We consider programs as sets of lines of code, denoted by L. Recall the definition of a combinatorial complex (Mustafa Hajij 2023):
A combinatorial complex is a triple: $\langle S, \chi, rk \rangle$ consisting of a set S, a subset of the power set $\chi \subset \mathcal{P}(S)$, and a function $rk : \chi \to \mathbb{N}$, called the rank function. The triple must satisfy the following axioms:

$$\forall s \in S,\ \{s\} \in \chi \tag{1}$$

$$s \subset s' \Rightarrow rk(s) \leq rk(s') \tag{2}$$

We take as S the set of all lines of code, L, in a source file. To construct χ, we consider functions as elements of $\mathcal{P}(S)$. To satisfy the axioms, we send each individual line $l \in L$ to the singleton set $\{l\} \in \chi$. All that remains is to construct the rank function, which we define recursively as follows:

$$\begin{aligned} rk(x \in \chi) &= 1 + max_{x' \subset x} rk(x') \\ rk(x) &= 0, \ \nexists\, x' \subset x \end{aligned} \tag{3}$$

It is trivial to prove this construction satisfies the required axiom of equation (2); simply consider any $x \subset x'$, then $rk(x) + 1 \leq rk(x')$ by definition. Unpacking the construction, we observe that functions which do not call other functions will have rank 0, while functions with complex sub-calls will have rank equal to the maximum called function rank plus 1. To avoid paradoxical results, we interpret functions to contain the lines of code which define their signature and end of scope.

To capture properties of the software beyond the single source file we are provided with, we introduce the concept of a *sub-function*. This is defined as (non-generic) calls to functions which reside outside of the source file. As an example, the outgoing arrows from "File B" in 1 can still be captured by our framework despite no knowledge of their destination. We consider such sub-functions to define the set $\chi' \subset \mathcal{P}(S)$ of cells to which assign rank 0. Intuitively, this allows our method to capture information provided by relationships between source files without needing to access to the details of those relationships.

Structural Measures Equipped with this framework, we now introduce a set of metrics of the combinatorial complexes arising from software. First, we consider the "height" of the complex, defined as:

$$H(L) = max_{x \in \chi} rk(x) \tag{4}$$

This serves as a natural extension of the notion of abstraction depth. Intuitively, higher values of H then indicate the presence of "longer", or "more complex", structural components of the software. Practically, this means that larger H indicates the presence of good "detail hiding", as taller complexes arise from software which leverages the power of abstraction.

Next, we consider the out-going "degree", which we define by:

$$|\chi'| \tag{5}$$

where χ' denotes the set of sub-function cells. This serves as an extension of the coupling/cohesion measures used existing works. Intuitively, source files with many out-going connections are correspondingly less self-contained which likely indicates poor cohesion.

To properly capture the relevant intra-file sturcture, we require the notion of adjacency in a combinatorial complex. We say that cells $x, y \in \chi$ are adjacent when:

$$\exists z \in \chi \textit{ such that } x \subset z \wedge y \subset z \wedge x \neq z \wedge y \neq z \tag{6}$$

similarly, $x, y \in \chi$ are co-adjacent when:

$$\exists z \in \chi \textit{ such that } z \subset x \wedge z \subset y \wedge x \neq z \wedge y \neq z \tag{7}$$

Such cells z are referred to as bridge cells. In this setting, adjacency encodes a "both are called by" ternary relation, and co-adacency encodes a "both call" relation, again, ternary. We note that these notions can be further specified to "k-up/down" versions by asserting things about the relative ranks of the cells involved. Observe that these intricate higher order relations permit the extraction of rich structural information that existing class-centric code analysis techniques cannot obtain. To construct a simple measure from these relations, we consider the graph-theoretic notion of volume of the $+1$ and -1 adjacency matrices for all ranks [2]. We recall that volume in this sense means the total number of relations normalized by the max possible relations. Concretely:

$$Vol(A_r) = \frac{\sum_{ij} A_r[i,j]}{|A_r|} \tag{8}$$

where A_r denotes the adjacency matrix for cells of rank r. These matrices are constructed in the apparent way. Intuitively, the volume measure captures a sense of how densely

[2]Recall that $+1$ adjacency refers to cells connected by a bridge cell of exactly 1 rank more. -1 adjacency is defined analogously.

the cells are connected, w.r.t higher and lower rank bridge cells.

Finally, we consider the number of "low" and "high" rank cells (R_L, R_H) in the complex. We define these as the cardinality of the sets induced by a partitioning of cells by the threshold $rk = 2$. Specifically:

$$R_L = |\{x \in \chi \mid rk(x) \leq 2\}| \tag{9}$$

$$R_L = |\{x \in \chi \mid rk(x) > 2\}| \tag{10}$$

The value of 2 was chosen because we observe that on average, real-world code exhibits a dramatic inflection point in rank distribution at $rk = 2$. We leverage all the notions discussed in this section to provide a comprehensive picture of the software structure.

Experiments

To conduct our empirical study, we use samples from the GitHub Code dataset [3] for analysis. To prevent the analysis of overly trivial pieces of code, we filter the dataset by a minimum size, defined by the total number of characters in the source file. This size is fixed at 25000. We sample a total of 10000 pieces of code from the dataset for each of the three languages: Java, C++, and Rust. We pre-process the data by throwing out files with less than 5 methods to ensure meaningful complexes are constructed. We designed and implemented a custom code parser in python which extracts the combinatorial complex structure discussed in the Methods section. Thanks to the lightweight nature of our analytical framework, parsing for all 30000 samples was completed in under 10 minutes.

To study the relative structural properties of the languages, we consider averages (across 10k samples) of the measures discussed above. This large sample size allows us to gain insights on how language features influence software structure. We present our results next.

Results

Baseline Analysis: Java vs. C++

	NOA (size)	WMC (complexity)	AHF (encapsulation)	CF (coupling)	LCC (cohesion)	DIT (inheritance)	PF (polymorphism)
Java	**1.76**	3.30	**0.82**	0.03	**94.28**	0.3	0.34
C++	1.86	**3.04**	0.15	**0.09**	74.92	**0.4**	0.35

Table 1: Results of OO metrics across Java and C++. The bolded values indicate that the metric is more desirable. If neither value is bolded, the metrics are comparable or represent a trade-off.

Table 1 provides the results from the baseline analytical framework. We observe that C++ exhibits higher coupling and lower cohesion than Java. Additionally, Java is better encapsulated at the cost of higher complexity. We now proceed with the results of our proposed analytical framework.

Proposed Method: Java vs. C++ vs. Rust

Sub-Functions and Encapsulation Figure 2 shows a plot of sub-functions (sub-cells, χ') vs. complex size $|\chi|$. Java's superior encapsulation is immediately observed from this, with significantly fewer coupling relations to outside code. Interestingly, we notice that for smaller complexes ($|\chi| \leq 30$) all languages are similar, but Java performs by far the best for larger complexes. This is a testament to the efficacy of our method, as it successfully recovers the same observation made by the baseline analytic framework. Beyond that, we do not observe much different between Rust and C++ on this metric. This aligns with intuition: C++ enforces fewer strict OO-rules than Java, making it easier to disobey encapsulation wisdom.

Complex Height and Abstraction Figure 3 shows complex height (H) vs. complex size. Similarly to Figure 2, we observe that Java and C++ exhibit comparable average complex height on smaller complexes; then C++ pulls ahead slightly on larger ones. This indicates that C++ tends to induce taller abstraction trees in large-scale code, albeit not by a very significant margin. The baseline analysis concurs with this conclusion, further demonstrating that our method extracts consistent findings.

Complex height also clearly highlights the difference between the OO and functional mindsets. The Rust code, on average, exhibits significantly shorter abstraction trees, indicating that choosing an OO language is indeed the superior choice if one desires "lazy" code with good detail hiding abilities. While the intuition that OO provides benefits to this affect is commonly held, it is exciting to see that this idea is both formally quantifiable and measurable in the real world.

Low and High Rank Cells Figures 4 and 5 show plots of low and high rank cells respectively. Yet again, the emerging trend of C++ and Java parting ways on larger complexes (of size around $40 - 50$) appears. We observe that on these larger complexes, Java tends to exhibit fewer lower rank cells than C++. This aligns with the baseline observation that Java code tends to be somewhat "more complex" than C++ code. Interestingly, while C++ and Rust are functionally equivalent in terms of low rank cells, C++ (alongside Java) contains far more high rank cells than Rust, for all complex sizes. This adds an additional perspective on the trend observed in Figure 3: OO languages are better at promoting the use of abstraction.

Distributional Measures Figures 6 and 7 show plots of numbers of cells and adjacency volumes vs. cell rank. This data reveals that on average, all three languages experience inflection points in cell rank distribution at $rk = 2$. This is interesting because it highlights an aspect of code structure which is common between OO and functional languages. One may consider what this means for software development: is there a universally optimal level of abstraction? While such a question is far beyond the scope of this work,

[3] GitHub Code Dataset link: https://huggingface.co/datasets/codeparrot/github-code

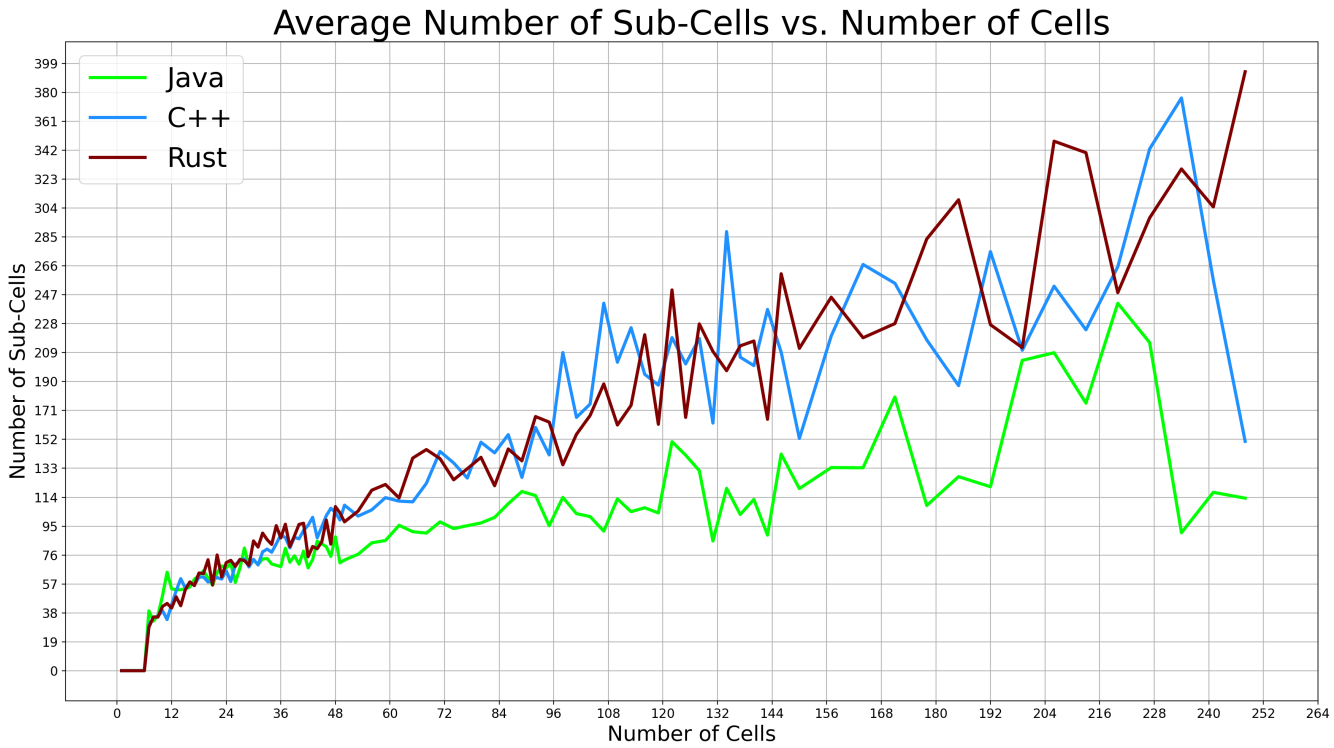

Figure 2: Average number of sub-cells plotted against complex size. Languages are color-coded. Java exhibits the lowest quantities, while C++ and Rust are not significantly different and exhibit noticeably increased values starting with complexes of approximately size 25.

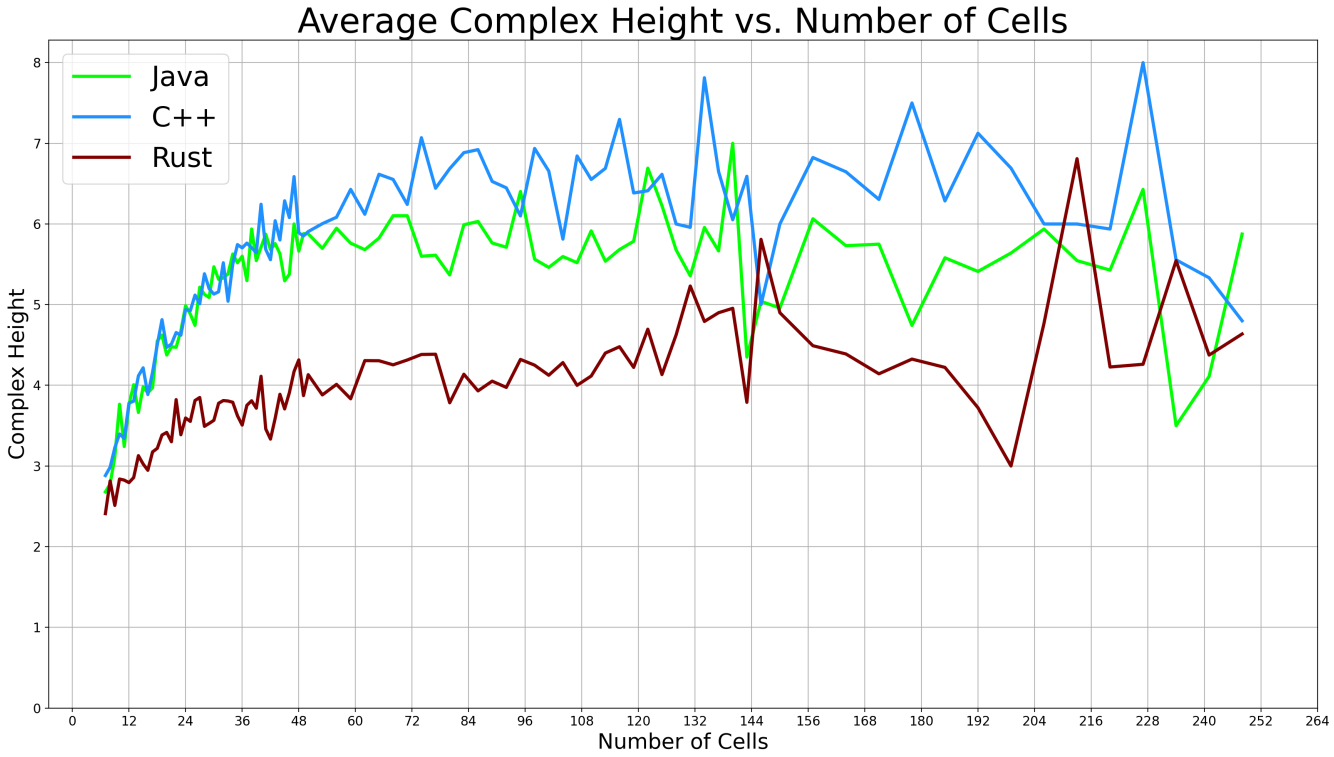

Figure 3: Average complex height plotted against complex size. Rust produces the shortest complexes across the board. C++ and Java are very similar on smaller complexes, but C++ tends to produce taller complexes when the size is larger.

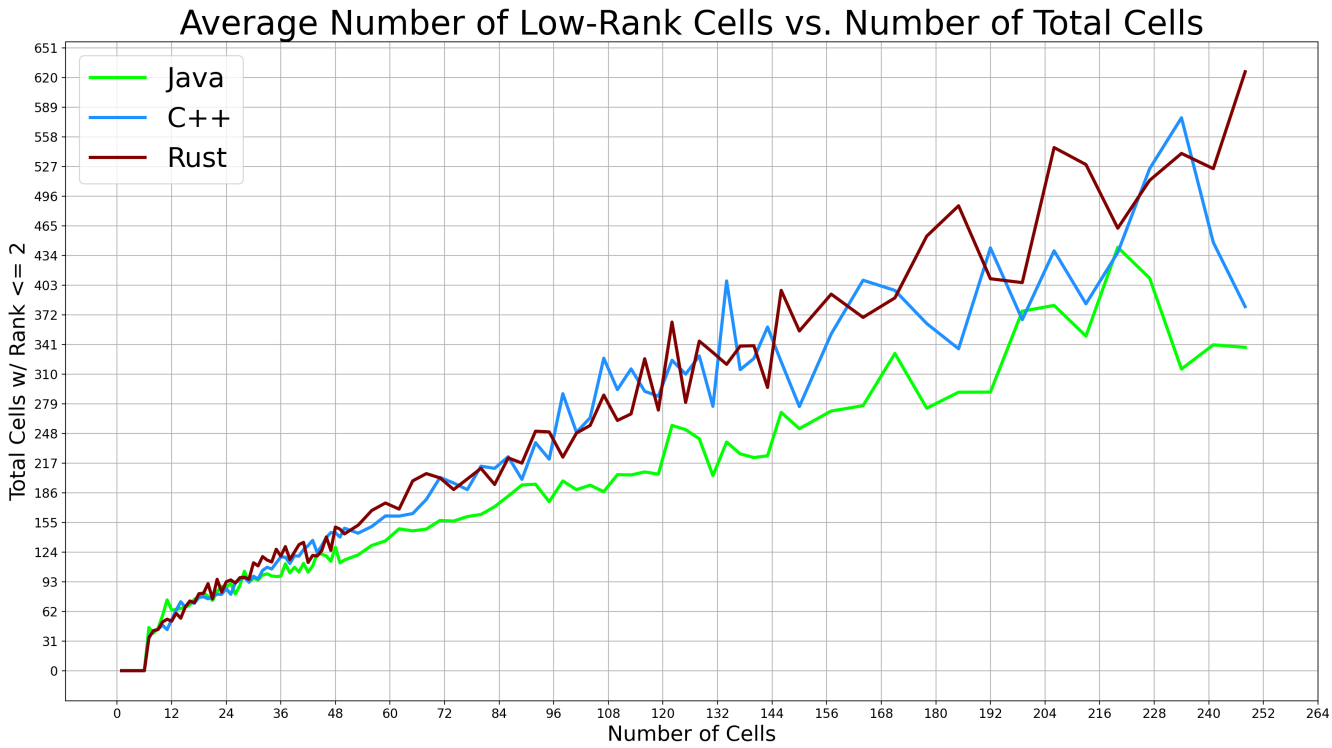

Figure 4: Average number of low rank cells vs. complex size. All languages are comparable to each other until complex size surpasses ~ 40; at which point Java falls behind C++ and Rust which remain similar.

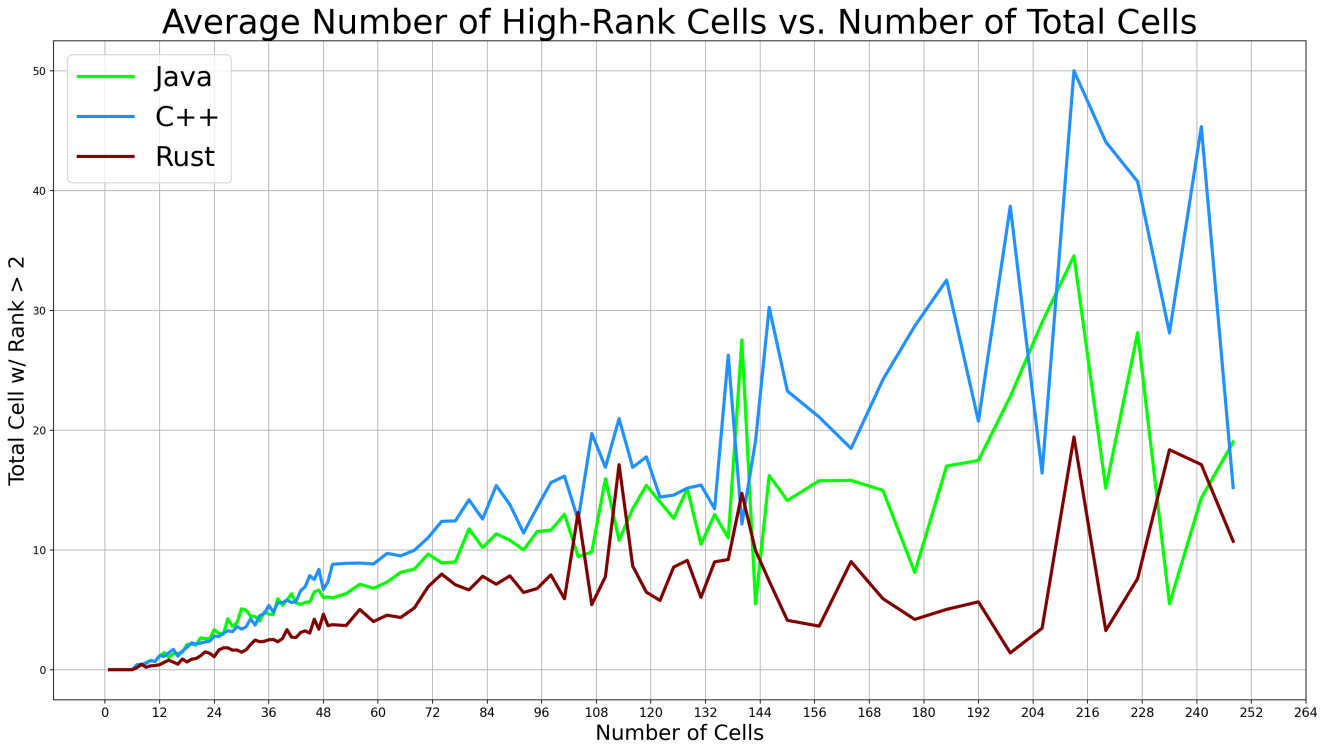

Figure 5: Average number of high rank cells vs. complex size. All languages are again comparable until complex size ~ 40. Here, C++ produces the most high rank cells, with Rust producing the least, and Java sandwiched in between.

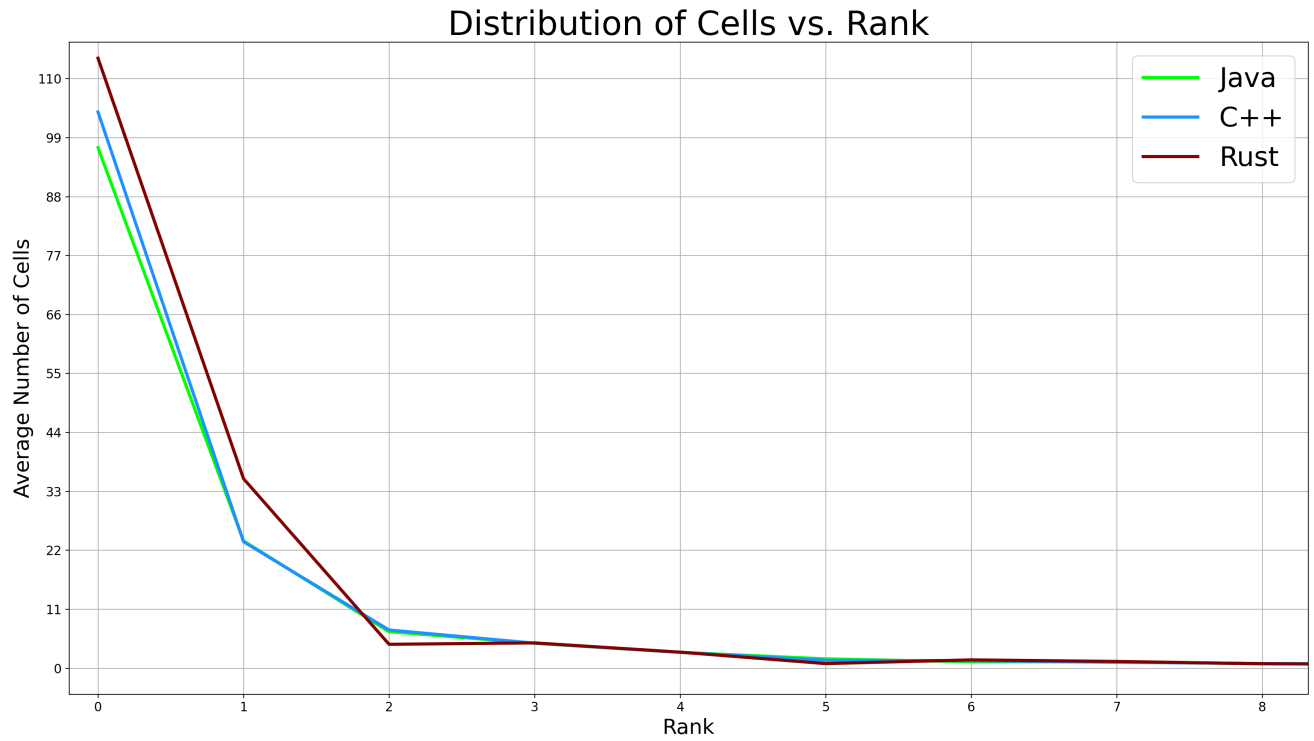

Figure 6: Distribution of cells over rank. All languages are effectively the same for rank 6 and onwards, with the most significant differences occurring at low ranks. Here, Rust exhibits higher values compared to C++ and Java.

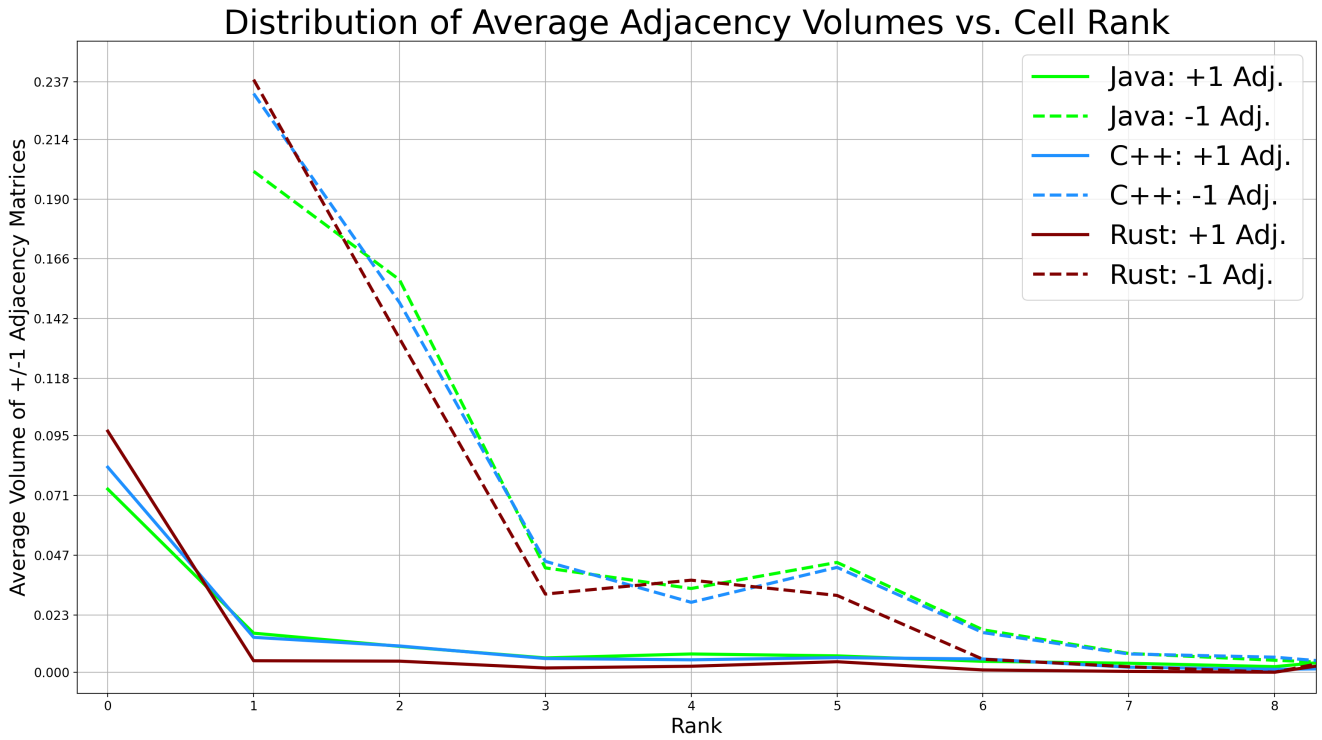

Figure 7: Average density of $+1$ and -1 adjacency matrices vs. cell rank. The OO languages exhibit higher up-adjacency densities at ranks above 0, while the down-adjacency densities are comparable across all languages until rank 5. Here, the OO languages exhibit higher values again.

we are excited that our proposed theoretical framework is powerful enough to permit the discussion such ideas.

Examining the adjacency volumes uncovers an interesting phenomenon: OO languages contain denser higher rank cell relationships than their functional counterparts. Moreover, this trend appears at $rk = 1$ for up-adjacency, but not until $rk = 5$ for down-adjacency. This means that OO languages "make better use of" lower rank cells because up-adjacency is defined as mutual caller ternary relations. Much like the observations on complex height and abstraction, this is a quantifiable benefit of adopting the OO paradigm. However, the trends observed here say something subtly stronger: not only do OO languages encourage good abstraction, but these abstractions actually get used more effectively as well. Fascinatingly, both Java and C++ exhibit the same degree of improvement over Rust, suggesting that this is a fundamental attribute of the OO philosophy itself.

Conclusion

In this paper, we introduced a novel software analysis framework which addresses many common limitations of existing techniques. We developed a theoretical foundation for cross-language intra-file dataset-scale code analysis; demonstrated it is consistent with prior analyses, and leveraged its superior analytic power to uncover new insights about why OO languages are preferred over functional ones. Our method establishes a foundation for formally understanding deeper questions about software, and we are excited to see what further discoveries it enables.

References

Awang Abu Bakar, N. 2016. The Analysis of Object-Oriented Metrics in C++ Programs. *Lecture Notes on Software Engineering*, 4: 48–52.

Ejiiogu, L. O. 1985. A Simple Measure of Software Complexity. In *Sigplan Notices, V20*.

Kumari, U.; and Bhasin, S. 2011. Application of Object-Oriented Metrics To C++ and Java: A Comparative Study. *SIGSOFT Softw. Eng. Notes*, 36(2): 1–10.

Mustafa Hajij, G. Z. K. N. R. T. B. M. T. S., Theodore Papamarkou. 2023.

Pan, W.; and Chai, C. 2017. Measuring software stability based on complex networks in software. In *Cluster Computing*.

Xinxin Xu, Y. L., Zengyou Zhang; and Wang, L. 2022. Topological Structure Analysis of Software Using Complex Network Theory. In *Mathematical Problems in Engineering*.

Yutao Ma, K. H.; and Du, D. 2005. A qualitative method for measuring the structural complexity of software systems based on complex networks. In *Asia-Pacific Software Engineering Conference*.