POLYSTAT: New Object Calculus to Improve Static Program Analysis

Yegor Bugayenko

April 28, 2021

Abstract

The quality of software, as one of the most important factors of business success in modern economy, may be increased by using static program analysis. The majority of modern programs are written in object-oriented programming languages, like Java or C++, while most static analysis methods treat them as procedural and algorithmic ones, reducing the ability to better understand programmers' intent and find more functional defects. The absence of a formal object calculus behind modern object-oriented languages is one of the key reasons. A development of such a calculus and using it in static analysis may significantly contribute to object-oriented programming as a whole and move static analysis to a principally new level.

1 Background

1.1 WHAT IS STATIC ANALYSIS? As was studied by Planning (2002), software bugs costed the U.S. economy about $59.5 billion annually. Static program analysis is a valuable part of software quality control and is performed without actually executing programs, as explained by S. Wagner et al. (2005) and Møller and Schwartzbach (2019). A growing commercial use of static analysis is in the verification of properties of software used in safety-critical computer systems and locating potentially vulnerable code, as noted by Chess and West (2007). According to Jackson and Rinard (2000), the importance of code analysis of all kinds will only grow in the future.

1.2 Usefulness Static analyzers suggest programmers to pay attention to certain locations in the source or binary code, highlight good candidates for fixing, and sometimes even suggest fixes. The final decision of whether to fix the code or leave it "as is" is made by the programmer. Very often the code is delivered to end-users without fixing the bugs found by analyzers, as noted by Steidl and Eder (2014). Also, as shown by Kremenek and Engler (2003), most warnings do not indicate real bugs. However, according to Sadowski et al. (2018), only a few percent of programmers react negatively to recommendations of analyzers.

1.3 Unsoundness Some functional bugs may be mistakenly reported where the code is actually correct or couldn't have led to significant misbehavior of the software (S. Wagner et al., 2005). Moreover, big proportion of "false positive" results in static analysis tools being disregarded by developers as was examined by Johnson et al. (2013). The inaccuracy due to "false positives" is known as "unsoundness" of analysis (Jackson and Rinard, 2000). It was explained by Møller and Schwartzbach (2019) that "a program analyzer is sound if it never gives incorrect results (but it may answer *maybe*)." Accuracy of 70% is considered to be high for modern analyzers, since ones that effectively find errors have "false positive" rates from 30% to 100% (Engler et al., 2000; Foster, Terauchi, and Aiken, 2002; Flanagan and Freund, 2000; D. A. Wagner et al., 2000). As noted by Gosain and Sharma (2015), "precise analysis is more costly."

1.4 Standards There are organizations, such as MISRA, SEI, MITRE and ISO, which publish guidelines, recommendations and standards for software developers (MISRA, 2012; CERT, 2016; ISO, 2011; RTCA, 2011; IEC, 2015b; IEC, 2015a; IEC, 2010). Some static analyzers take those recommendations into account and get certified, for example, Coverity, Klocwork and CodeSonar are such analyzers certified by TüV SüD.

1.5 Front-End Most analyzers have three major components of their architecture, as described by Binkley (2007): the parser, the internal representation, and the analysis of this representation. The parser, which usually includes lexer, preprocessor, and tokenizer (together known as "front-end") in most analyzers is language-specific: it can only parse a source code written in one or a few programming languages. Some analyzers are polyglots, which usually is realized via an intermediary language (also known as "intermediary representation" or IR), which the source language is translated into, before the analysis is done. For example, Phasar (Schubert, Hermann, and Bodden, 2019), CodeChecker (Hungary, 2019) and Clang Static Analyzer (Lattner, 2020) are open source analyzers of such kind—they analyze LLVM code (Lattner and Adve, 2004), which may be generated from C/C++, Java, C# and some other languages.

1.6 Internal Representations As mentioned by Binkley (2007), "there are almost as many internal representations as there are source-code analyses." Some classic examples include the Control-Flow Graph (CFG), the Call Graph (CG), and the Abstract Syntax Tree (AST).

1.7 Back-End The analysis itself (also known as "back-end"), as suggested by Binkley (2007), can be classified along six dimensions: static versus dynamic, sound versus unsound, safe versus unsafe, flow sensitive versus flow insensitive, context sensitive versus context insensitive, and complexity. Gosain and Sharma (2015) gives a detailed summary of methods and techniques used by existing analyzers to find bugs, to name just a few: path-sensitive data flow analysis (Kremenek, 2008), alias analysis (Wu et al., 2013), type analysis (Wand, 1987), symbolic execution (Slaby, 2013), value flow analysis (Sui and Xue, 2016), abstract interpretation (Slaby, 2013), control flow analysis (Allen, 1970), pointer analysis (Smaragdakis and Balatsouras, 2015), theorem proving (Darvas, Hähnle, and Sands, 2005), constraint-based analysis (Suter, 2014), summary-based pointer analysis (Buss, 2008), and so on.

2 Types of Bugs

2.1 Maintainability Bugs Bugs detected by analyzers are either maintainability, functional, or security. Maintainability bugs, also sometimes referred to as "code style violations," won't cause any functional issues to the product, if left unfixed. They may not even be considered as bugs by some developers, since the maintainability itself is a vague term, as explained by Broy, Deissenboeck, and Pizka (2006). However, in most cases, fixing them makes the source code more readable, which indirectly leads to reducing the amount of logical mistakes made by programmers when maintaining at later date (Posnett, Hindle, and Devanbu, 2011). Here is an example of Java maintainability bug detectable by the `AssignmentInOperand` "check" of PMD analyzer; the code sample is adopted from the book of Sierra and Bates (2005, p.475):

```
char[] buffer = new char[1024];
int pos = 0;
while ((data = reader.read()) > 0) {
  buffer[pos++] = (char) data;
}
```

Here, the variable `data` is assigned and at the same time used as an operand for the `while` statement. Such a construct may confuse programmers, especially less experienced ones; it must be avoided, according to the command–query separation principle suggested by Meyer (1997). In order to increase readability of the code it may be refactored (Fowler, 2018) to decrease complexity, coupling, and other qualities (Yamashita and Moonen, 2012):

```
char[] buffer = new char[1024];
int pos = 0;
while (true) {
  int data = reader.read();
  if (data < 0) {
    break;
  }
  buffer[pos++] = (char) data;
}
```

2.2 Functional Bugs Functional bugs are more difficult to find, but they are more important, because they may cause runtime errors if left unfixed. Even after the refactoring suggested, the Java code snippet mentioned above contains a functional bug related to a possible buffer overflow, if the number of symbols in the reader is bigger than 1024. A possible fix would look like this:

```java
char[] buffer = new char[1024];
int pos = 0;
while (true) {
  int data = reader.read();
  if (data < 0) {
    break;
  }
  if (pos >= buffer.length) {
    throw new RuntimeException("Too much data");
  }
  buffer[pos++] = (char) data;
}
```

This is not a perfect fix though. One may argue that it does not improve the code in any way, since the exception it throws is an unspecific runtime exception as opposed to the semantically more meaningful out-of-bounds exception thrown in the previous example. A more meaningful fix would require more code refactoring most probably in other places of the source code, to prevent buffer overflow from happening.

2.3 Security Bugs Security bugs (aka “security vulnerabilities”) don’t affect the functionality of a system, but may cause leakage of sensitive data or its unauthorized modification, which is usually prohibited by the non-functional part of requirements documentation. The snippet above reads the data into memory and raises exception in case of buffer overflow, while the data remains in memory. If the data is sensitive and the application crashes right after the exception is raised, the memory will contain the data open for exposure. Here is how this problem could be fixed:

```java
char[] buffer = new char[1024];
int pos = 0;
while (true) {
  int data = reader.read();
  if (data < 0) {
    break;
  }
  if (pos >= buffer.length) {
    Arrays.fill(buffer, 0); // Here!
    throw new RuntimeException("Too much data");
  }
  buffer[pos++] = (char) data;
}
```

It's important to mention that as a result of simple analysis the code above grew up from five lines to 13 lines. This is a well-known effect of static analysis: the code gets bigger, while its quality increases.

2.4 Object-Oriented Bugs It is possible to classify functional bugs as either common or object-oriented specific. The bug suggested above belongs to the first category and may exist in many languages, including those that don't have object-oriented features, for example C. In the example above the method `read()` is used to read the data. The method belongs to the object `reader`, whic most probably is an instance of the abstract class `java.io.Reader`. This class also has method `close()`, which has to be called when reading is finished: it's a conventional Java agreement noticable by the presence of the `Closeable` interface in the list of parents of the class `Reader`. The code above doesn't call `close()`, which may lead to resource leakage and eventual program crash. The code may be fixed like this (`try`/`finally` statements are added):

```
char[] buffer = new char[1024];
int pos = 0;
try {
  while (true) {
    int data = reader.read();
    if (data < 0) {
      break;
    }
    if (pos >= buffer.length) {
      Arrays.fill(buffer, 0); // Here!
      throw new RuntimeException("Too much data");
    }
    buffer[pos++] = (char) data;
  }
} finally {
  reader.close();
}
```

There are other bugs related solely to violations of object design. There is a short list of most obvious ones, while the full list is yet to be determined:

- Resource leakage due to violation of contracts (already explained);
- Fragile base class problem (Mikhajlov and Sekerinski, 1998);
- The “diamond problem” due to wrong inheritance (Roebuck, 2011);
- Type mismatches in dynamically and/or weakly typed languages;
- Memory leaks through static variables/methods abuse;
- Non-intentional data serialization;
- Broken equality relationship due to subtyping (Sarcar, 2020);
- Name clashes due to namespace borders violation;
- Missed initialization of parent class (Roebuck, 2011);
- Object comparison by identity instead of value (Bloch, 2016);
- Spaghetti inheritance problems (Geetha et al., 2008);
- Circle-ellipse problem (Majorinc, 1998);
- Stack overflow due to circular dependencies;
- Accidental calls to base class due to incomplete method overloading;
- Data precision losses at boxing/unboxing (Bloch, 2016);
- Concurrency side-effects in mutable objects (Goetz et al., 2006);
- Identity mutability problem, especially in hash maps (Goetz et al., 2006);
- Class casting errors.

It's important to mention that not all bugs are easily fixable and very often may remain in the code even after static analysis discovers them—simply because programmers may not have enough time and/or skills to fix them.

3 Problem Formulation

3.1 Procedural Analysis Even though existing methods of static analysis work with object-oriented (OO) languages, like Java, C++, C#, Python, and JavaScript, they treat the source code and its constructs as if they were written in imperative procedural languages, like ALGOL and Assembly. They tend to ignore the complexity of "object-orientedness" (inheritance, generics, method overloading, annotations, and so on) and deal with low-level statements and operators. Polyglot analyzers even map the original language to a more primitive intermediate representation, losing the entire semantic of objects and the original intent of programmers. For example, LLVM, which is used in many modern analyzers, doesn't have a notion of an object or a class. The original object-oriented code is translated into LLVM and then is analyzed as a collection of LLVM IR instructions (very close and similar to Assembly). It seems that this design of existing analyzers is motivated by the absence of a common OOP formalism, which was explained later in the Section 3.3.

3.2 Redundant Completeness LLVM, as well as GraalVM (Würthinger et al., 2013), is a powerful instrument to enable cross-platform compatibility between programming languages (Lattner and Adve, 2004). It usually does its job in four steps: 1) converts C++ source code to LLVM IR instructions, 2) LLVM IR to Bitcode, 3) Bitcode to x86 Assembly, 4) Assembly to native binary. Analyzers work either with LLVM instructions or with the Bitcode, parsing them into an AST or CFG, then traversing and reasoning. LLVM, being a Virtual Machine (VM), is very much concerned about executability of the source code after it gets to the native binary. That's why the instructions produced at the first step are "complete": they include everything required for a target platform to run the code. For example, LLVM doesn't have objects and Garbage Collector (GC), while Java programs expect this feature to be present in the VM. Thus, a Java to LLVM mapper has to add a GC into the LLVM code it generates, on top of the Java code being mapped. A classic "Hello, world!" Java example with a single class, one statement, and five lines of code would produce dozens of thousands of LLVM lines of code[1]. Such a redundancy makes static analysis more difficult, since it's necessary to filter out "boilerplate" code, which is important at runtime, but absolutely useless at the time of analysis.

3.3 Lack of OOP Formalism Static analyzers for OO code are designed in ad hoc way mostly because there is no formal theory of object-oriented programming (OOP). Stefik and Bobrow (1985): "The term has been used to mean different things." Madsen and Møller-Pedersen (1988): "There are as many definitions of OOP as there are papers and books on the topic." Armstrong (2006): "When reviewing the body of work on OO development, most authors simply suggest a set of concepts that characterize OO, and move on with their research or discussion. Thus, they are either taking for granted that the concepts are known or implicitly acknowledging that a universal set of concepts does not exist." Nierstrasz (1989): "There is no uniformity or an agreement on the set of features and mechanisms that belong in an OO language as the paradigm itself is far too general." It's hard to say why exactly a very mature domain of OOP still doesn't have a unified formal ground, while others do, including functional and logical programming. Most probably this happens due to intensive industry support given to major programming languages by different large tech companies: Oracle supports Java, Microsoft supports C#, Apple supports Swift and Objective-C, and so on. They can't agree to each other, while programmers demand new features to be introduced. However, it is merely a guess.

3.4 OOP Defects to Discover New object calculus will provide the ability to analyze OO code directly, without converting it to lower-level procedural instructions and losing OO semantics. Thanks to this, it will be possible to detect functionality defects which were not detectable before (or difficult to detect), including but not limited to: invalid inheritance loops; un-initialized class and object attributes; eager execution during object construction; hidden inter-object and inter-class dependencies; and many more.

3.5 Theoretical Problem The inability to formally specify OO programs using existing mathematical apparatus, leads to low effectiveness of code analysis instruments such as static analyzers and compilers, which causes more functional

[1] `https://github.com/yegor256/llvm-playground`

defects in software products, which causes customer frustration and financial losses for the business.

4 Prior Art

4.1 Existing Products There are many static analyzers on the market: free and open source tools such as Checkstyle for the code written in Java (Burn, 2020), PMD for Java (PMD, 2020), FindBugs for Java (Ayewah et al., 2008), cpplint for C/C++ (Kruse, 2020), ESLint for JavaScript (Zakas, 2020), Rubocop for Ruby (Batsov, 2020), Flake8 for Python (Stapleton, 2020), and others (Rutar, Almazan, and Foster, 2004); commercial tools such as Coverity for C, C++, Java, and many other languages (*Coverity* 2019); Klocwork for C, C++ C#, and Java (*Klocwork* 2020); PVS-Studio for C, C++ C#, and Java (*PVS-Studio* 2020); and others.

4.2 Object Theories Earlier attempts were made to formalize OOP and introduce object calculus, for example imperative calculus of objects by Abadi and Cardelli (1995), Featherweight Java by Igarashi, Pierce, and Wadler (2001), Larch/C++ by Cheon and Leavens (1994), Object-Z by Duke et al. (1991) and VDM++ by Durr and Van Katwijk (1992). However, all these theoretical attempts to formalize object-oriented languages were not able to fully describe their features, as for example was noted by Nierstrasz (1991): "The development of concurrent object-based programming languages has suffered from the lack of any generally accepted formal foundations for defining their semantics." In addition, when describing the attempts of formalization, Eden (2002) summarized: "Not one of the notations is defined formally, nor provided with denotational semantics, nor founded on axiomatic semantics." Moreover, despite these efforts, Ciaffaglione, Liquori, and Miculan (2003b), Ciaffaglione, Liquori, and Miculan (2003a), and Ciaffaglione, Liquori, and Miculan (2007) noted in their series of works that a relatively little formal work has been carried out on object-based languages and it remains true to this day.

4.3 Our Prior Results Earlier successful attempts have already been made by the authors to improve and formalize OOP: *Elegant Objects*, a series of books, were published (Bugayenko, 2016; Bugayenko, 2017) where traditional OO design concepts were criticized (alternatives were suggested too), including NULL references, static methods, getters, mutability, and so on; Takes[1], Cactoos[2], and 20+ other Java/Python/C# frameworks were developed by open source volunteers with said new design concepts in mind[3], demonstrating better readability of the code; EOLANG, an experimental programming language, developed by a group of 10+ open source volunteers[4], also demonstrated advantages of new design principles, comparing to traditional OOP languages like Java and C++; a few academic papers have been published on the subject (Bugayenko, 2020), demonstrating benefits of a new OOP paradigm.

5 Method

5.1 Research Objectives The following goals seem reasonable to achieve:

- Analyze most popular existing object-oriented languages and identify their similarities and differences;
- Formalize OOP similar to how functional and logical programming are formalized and introduce *object calculus*, similar to λ-calculus (Church, 1932);
- Create a new *programming language* based on the new object calculus;
- Create a set of automated *mappers* from modern languages like Java and C++ to the new language (only a few mappers will be created by the authors, while others will be contributed by the open source community spending 2-3 staff-months on each of them, similar to how it works with LLVM mappers);
- Introduce new *static analysis methods* based on formal reasoning on the new object calculus, partially inheriting existing methods of procedural static analysis.

5.2 Key Challenges The most important research questions to be answered:

- Is it possible to define a single IR that is able to represent all the different characteristics of OOP languages?
- What is the optimal set of terms and operations of an object calculus, which would enable representation of any possible object model?

[1]`https://www.takes.org`
[2]`https://www.cactoos.org`
[3]`https://www.elegantobjects.org`
[4]`https://github.com/yegor256/eo`

- How existing OOP constructs can be mapped to a new object calculus without losing their functionality?
- How semi-OO languages (like Python or JavaScript) can be mapped on a new strict object calculus?
- How new object calculus can be mapped to existing programming?
- Is it possible to identify defects through formal reasoning on new calculus?

5.3 Evaluation Criteria Although there is no common formula or benchmark methodology of assessing the quality of analyzers (Delaitre et al., 2015), many researches use "recall" and "precision" metrics as a starting point (Nunes et al., 2017; Lu et al., 2005; Meade, 2012; Kupsch and Miller, 2009; Delaitre et al., 2015), where the former measures the proportion of "true positives" identified by analyzer compared to the total number of defects known to be existing in the code, and the latter indicates the trustworthiness of the tool by measuring the proportion of correct warnings to the total number of warnings detected by an analyzer. It is expected to gain at least 10% on each metric, in comparison with the latest version of Clang Static Analyzer (CSA). It is also important to take into account the time of analysis, which must not be much longer than what CSA spends to analyze a program of similar size.

5.4 Verification In order to verify research results it is suggested to apply the following testing procedure: 1) take a hundred projects from GitHub, which have 1000+ stars and 100+ forks, and are labeled as C++ repositories; 2) filter our `.cpp` files with less than 50 and more than 500 NCSS; 3) analyze them with Clang Static Analyzer (baseline); 4) analyze them using the methods developed during the research (output); 5) manually review defects found in the baseline and absent in the output, decreasing "recall"; 6) manually review in the opposite direction and decrease "precision"; 7) stop reviewing after 1000 files or when either recall or precision fail to satisfy the evaluation criteria.

6 Proposed Solution

6.1 New Principles How exactly the new object calculus will look is to be determined by the research, but the following principles seem to be reasonable to address: unlike subtyping, *inheritance* is error-prone method of code reuse[1]; *pointers* and *NULL references* don't belong to OOP[2]; objects are the only first-class citizens, while *classes* are redundant code templates; *static* methods and static attributes negatively affect maintainability[3]; *mutability* leads to object over-sizing and feature creep[4]; runtime *reflection* on types is a design smell[5] and a threat to its consistency. More details are available in the *EOLANG: Object-Oriented Programming Language and Object Calculus* paper by Yegor Bugayenko (to be submitted to one of the journals on programming languages).

6.2 Static Analysis The proposed EOLANG programming language may be used as intermediary representation for static analysis of OOP code. Using EOLANG and φ-calculus behind it, may enable higher precision and accuracy in finding defects in OOP code, especially in C++ and Java. More details are available in the *Using EOLANG for Finding Defects in Object-Oriented Programs* paper by Yegor Bugayenko (to be submitted later to a conference).

6.3 Architecture The Figure 1 explains the architecture. First, the source code in Java, C++, Python, or almost any other programming language is translated to EOLANG by one of available open source Mappers. Most of them will most likely be written in Java with the help of ANTLR4 (Parr, 2013). Then, EOLANG is sent to the Sourcer, which creates XML instructions and modifies the IR. Also, the source code in, for example, C++ may be sent to the Parser, which will take some important parts of it, like inline comments or code formatting details, and also modify EOLANG objects accordingly. After that, Advisers query EOLANG objects and make modifications to it. Each Adviser may implement its own analysis method and enrich the IR with the relevant information. Finally, Rules step it and read the IR, trying to find bugs.

[1] https://www.infoworld.com/article/2073649/why-extends-is-evil.html
[2] https://www.yegor256.com/2014/05/13/why-null-is-bad.html
[3] https://codeburst.io/af3e73bd29dd
[4] https://www.yegor256.com/2014/06/09/objects-should-be-immutable.html
[5] https://wiki.c2.com/?RuntimeReflectionIsaDesignSmell

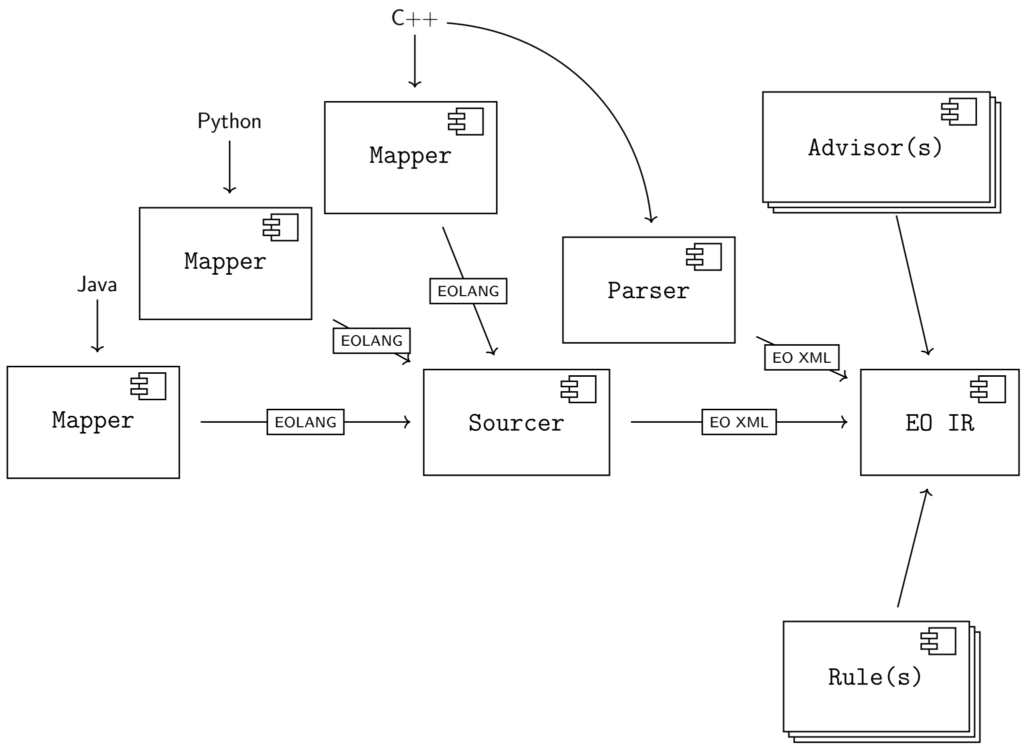

Figure 1: UML Component Diagram of key elements wired together

7 Expected Outcomes

7.1 Scientific Effect If the suggested research plan succeeds, the following positive outcome is possible for computer science and the domain of programming languages in particular:

- OOP will finally obtain its lacking component—object calculus;
- It will help design better languages, compilers, and static analyzers;
- New and better methods of static analysis will be introduced.

7.2 Industry Effect The industry of software development may benefit too:

- The quality of software will be increased, meaning less functional bugs (there are many other quality aspects of software to be increased, but this one is the most obvious and measurable);
- New programming language may become an alternative to Java and C++;
- It may be used in microservices, embedded software, and so on;
- New polyglot static analyzer may outperform CSA with better soundness and accuracy by at least 10%, as mentioned in the Section 5.3;
- The maintainability and security of the source code written by millions of programmers may be improved through the introduction of a common formal ground of OOP;
- The analyzer may be open-sourced and certified to help software teams deliver higher quality of code with no additional costs.

7.3 Probability of Success Even though, as was demonstrated above, many attempts to formalize object-oriented programming were not successful, we believe that our new research may produce the outcome we are looking for. First, as Section 4 explained, our existing tools, libraries, and frameworks empirically demonstrated the solidness of the new paradigm. Second, modern higher-level programming languages like Kotlin and Groovy, are much further away from low-level procedural paradigm, which dominated when first attempts to formalize OOP were made—this situation gives us higher chances to succeed. Third, the first version of our new object calculus has already been created and a new programming language on top it was implemented—this proof-of-concept, if future research is done properly, is a promising indicator of success.

References

Abadi, Martín and Luca Cardelli (1995). "An Imperative Object Calculus". In: *Theory and Practice of Object Systems* 1.3.

Allen, Frances E (1970). "Control Flow Analysis". In: *ACM SIGPLAN Notices* 5.7, pp. 1–19.

Armstrong, Deborah J (2006). "The Quarks of Object-Oriented Development". In: *Communications of the ACM* 49.2.

Ayewah, Nathaniel et al. (2008). "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5, pp. 22–29.

Batsov, Bozhidar (2020). *Rubocop*. URL: https://github.com/rubocop-hq/rubocop.

Binkley, David (2007). "Source Code Analysis: A Road Map". In: *Future of Software Engineering*. IEEE, pp. 104–119.

Bloch, Joshua (2016). *Effective Java*. Pearson Education India.

Broy, Manfred, Florian Deissenboeck, and Markus Pizka (2006). "Demystifying Maintainability". In: *Proceedings of the International Workshop on Software Quality*, pp. 21–26.

Bugayenko, Yegor (2016). *Elegant Objects*. Vol. 1. Amazon.

— (2017). *Elegant Objects*. Vol. 2. Amazon.

— (2020). "The Impact of Constructors on the Validity of Class Cohesion Metrics". In: *IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, pp. 67–70.

Burn, Oliver (2020). *Checkstyle*. URL: http://checkstyle.sourceforge.net/.

Buss, Marcio (Dec. 2008). "Summary-based Pointer Analysis Framework for Modular Bug Finding". In.

CERT (2016). *SEI CERT C Coding Standard – Rules for Developing Safe, Reliable, and Secure Systems*.

Cheon, Yoonsik and Gary T Leavens (1994). "A Quick Overview of Larch/C++". In.

Chess, Brian and Jacob West (2007). *Secure Programming with Static Analysis*. Pearson Education.

Church, Alonzo (1932). "A set of postulates for the foundation of logic". In: *Annals of Mathematics*.

Ciaffaglione, Alberto, Luigi Liquori, and Marino Miculan (Jan. 2003a). "Imperative Object-Based Calculi in Co-inductive Type Theories". In.

— (Aug. 2003b). "Reasoning on an Imperative Object-based Calculus in Higher Order Abstract Syntax". In.

— (2007). "Reasoning about Object-Based Calculi in (Co)Inductive Type Theory and the Theory of Contexts". In: *Journal of Automated Reasoning*.

Coverity (2019). URL: https://scan.coverity.com/.

Darvas, Ádám, Reiner Hähnle, and David Sands (2005). "A Theorem Proving Approach to Analysis of Secure Information Flow". In: *International Conference on Security in Pervasive Computing*. Springer, pp. 193–209.

Delaitre, Aurelien et al. (2015). "Evaluating Bug Finders–Test and Measurement of Static Code Analyzers". In: *IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems*. IEEE, pp. 14–20.

Duke, Roger et al. (1991). "The Object-Z Specification Language". In.

Durr, Eugene and Jan Van Katwijk (1992). "VDM++, A Formal Specification Language for Object-Oriented Designs". In: *Proceedings Computer Systems and Software Engineering*.

Eden, Amnon (2002). "A Visual Formalism for Object-Oriented Architecture". In: *Proceedings, Integrated Design and Process Technology*.

Engler, Dawson et al. (2000). "Checking System Rules Using System-specific, Programmer-written Compiler Extensions". In: *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*. Vol. 4.

Flanagan, Cormac and Stephen N. Freund (2000). "Type-based Race Detection for Java". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, pp. 219–232.

Foster, Jeffrey S, Tachio Terauchi, and Alex Aiken (2002). "Flow-sensitive Type Qualifiers". In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 1–12.

Fowler, Martin (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

Geetha, B.G. et al. (Jan. 2008). "A Tool for Testing of Inheritance Related Bugs in Object Oriented Software". In: *Journal of Computer Science* 4.

Goetz, Brian et al. (2006). *Java Concurrency in Practice*. Pearson Education.

Gosain, Anjana and Ganga Sharma (2015). "Static Analysis: A Survey of Techniques and Tools". In: *Intelligent Computing and Applications*. Springer, pp. 581–591.

Hungary, Ericsson (2019). *CodeChecker, a Static Analysis Infrastructure Built on the LLVM/Clang Static Analyzer Toolchain, Replacing Scan-build in a Linux or macOS (OS X) Development Environment*. URL: `https://codechecker.readthedocs.io/en/latest/`.

IEC (2010). *IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*.

— (2015a). *IEC 62279, Railway Applications - Communication, Signalling and Processing Systems - Software for Railway Control and Protection Systems*.

— (2015b). *IEC 62304, Medical Device Software - Software Life Cycle Processes*.

Igarashi, Atsushi, Benjamin C Pierce, and Philip Wadler (2001). "Featherweight Java: a Minimal Core Calculus for Java and GJ". In: *ACM Transactions on Programming Languages and Systems* 23.3.

ISO (2011). *ISO 26262: Road vehicles—Functional safety*.

Jackson, Daniel and Martin Rinard (2000). "Software Analysis: A Roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*, pp. 133–145.

Johnson, Brittany et al. (2013). "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" In: *2013 35th International Conference on Software Engineering*. IEEE, pp. 672–681.

Klocwork (2020). Perforce. URL: `https://www.perforce.com/products/klocwork`.

Kremenek, Ted (2008). *Finding Software Bugs With the Clang Static Analyzer*. URL: `https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf`.

Kremenek, Ted and Dawson Engler (2003). "Z-ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations". In: *International Static Analysis Symposium*. Springer, pp. 295–315.

Kruse, Thibault (2020). *cpplint*. URL: `https://github.com/cpplint/cpplint`.

Kupsch, James A and Barton P Miller (2009). "Manual vs. Automated Vulnerability Assessment: A Case Study". In: *First International Workshop on Managing Insider Security Threats*, pp. 83–97.

Lattner, Chris (2020). *Clang: a C Language Family Frontend for LLVM*. URL: `https://clang.llvm.org/index.html`.

Lattner, Chris and Vikram Adve (2004). "LLVM: A Compilation Framework for

Lifelong Program Analysis & Transformation”. In: *International Symposium on Code Generation and Optimization*. IEEE, pp. 75–86.

Lu, Shan et al. (2005). “Bugbench: Benchmarks for Evaluating Bug Detection Tools”. In: *Workshop on the Evaluation of Software Defect Detection Tools*. Vol. 5.

Madsen, Ole Lehrmann and Birger Møller-Pedersen (1988). “What Object-Oriented Programming May Be-and What It Does Not Have to Be”. In: *European Conference on Object-Oriented Programming*.

Majorinc, Kazimir (1998). “Elipse-Circle Dilemma and Inverse Inheritance”. In: *ITI 98*.

Meade (2012). *CAS Static Analysis Tool Study - Methodology*. Center for Assured Software, National Security Agency. URL: https://samate.nist.gov/docs/CAS%202012%20Static%20Analysis%20Tool%20Study%20Methodology.pdf.

Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Vol. 2. Prentice hall Englewood Cliffs.

Mikhajlov, Leonid and Emil Sekerinski (1998). “A Study of the Fragile Base Class Problem”. In: *European Conference on Object-Oriented Programming*.

MISRA (2012). *MISRA C:2012*.

Møller, Anders and Michael I Schwartzbach (2019). “Static Program Analysis”. In.

Nierstrasz, Oscar (1989). *A Survey of Object-Oriented Concepts*.

— (1991). “Towards an Object Calculus”. In: *European Conference on Object-Oriented Programming*.

Nunes, Paulo et al. (2017). “On Combining Diverse Static Analysis Tools for Web Security: An Empirical Study”. In: *13th European Sependable Computing Conference*. IEEE, pp. 121–128.

Parr, Terence (2013). *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf.

Planning, Strategic (2002). “The Economic Impacts of Inadequate Infrastructure for Software Testing”. In: *National Institute of Standards and Technology*.

PMD (2020). *PMD*. URL: https://pmd.github.io/.

Posnett, Daryl, Abram Hindle, and Premkumar Devanbu (2011). “A Simpler Model of Software Readability”. In: *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp. 73–82.

PVS-Studio (2020). PVS-Studio Team. URL: https://www.viva64.com/en/pvs-studio/.

Roebuck, Kevin (2011). *Object-Oriented Analysis and Design: High-Impact Strategies*. Tebbo.

RTCA (2011). *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*.

Rutar, Nick, Christian B Almazan, and Jeffrey S Foster (2004). “A Comparison of Bug Finding Tools for Java”. In: *15th International Symposium on Software Reliability Engineering*. IEEE, pp. 245–256.

Sadowski, Caitlin et al. (2018). “Lessons from Building Static Analysis Tools at Google”. In: *Communications of the ACM* 61.4, pp. 58–66.

Sarcar, Vaskaran (2020). *Quick Recap of OOP Principles*. Berkeley, CA: Apress.

Schubert, Philipp Dominik, Ben Hermann, and Eric Bodden (2019). “PhASAR: An Inter-procedural Static Analysis Framework for C/C++”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, pp. 393–410.

Sierra, Kathy and Bert Bates (2005). *Head First Java: A Brain-Friendly Guide*. O’Reilly Media.

Slaby, Jiri (2013). “Automatic Bug-finding Techniques for Large Software Projects”. In.

Smaragdakis, Yannis and George Balatsouras (2015). “Pointer Analysis”. In: *Foundations and Trends in Programming Languages* 2.1, pp. 1–69.

Stapleton, Ian (2020). *Flake8*. URL: `https://flake8.pycqa.org/en/latest/`.

Stefik, Mark and Daniel G Bobrow (1985). “Object-Oriented Programming: Themes and Variations”. In: *AI Magazine* 6.4.

Steidl, Daniela and Sebastian Eder (2014). “Prioritizing Maintainability Defects Based on Refactoring Recommendations”. In: *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 168–176.

Sui, Yulei and Jingling Xue (Mar. 2016). “SVF: Interprocedural Static Salue-flow Analysis in LLVM”. In: pp. 265–266.

Suter, Toni (2014). “Constraint Based Analysis”. In.

Wagner, David A et al. (2000). “A First Step towards Automated Detection of Buffer Overrun Vulnerabilities”. In: *NDSS*. Vol. 20. 0.

Wagner, Stefan et al. (2005). “Comparing Bug Finding Tools With Reviews and Tests”. In: *IFIP International Conference on Testing of Communicating Systems*. Springer, pp. 40–55.

Wand, Mitchell (1987). “A Simple Algorithm and Proof for Type Inference”. In: *Fundamenta Informaticae* 10.2, pp. 115–121.

Wu, Jingyue et al. (2013). “Effective Dynamic Detection of Alias Analysis Errors”. In: *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, pp. 279–289.

Würthinger, Thomas et al. (2013). “One VM to Rule Them All”. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pp. 187–204.

Yamashita, Aiko and Leon Moonen (2012). “Do Code Smells Reflect Important Maintainability Aspects?” In: *28th International Conference on Software Maintenance*. IEEE, pp. 306–315.

Zakas, Nicholas C. (2020). *ESLint*. URL: `https://eslint.org/`.