Eolang Analysis
Technical Report Stage I

HSE team
hsalekh@hse.ru
HSE
Moscow,Russia

Abstract

Object-oriented programming (OOP) is one of the most popular programming paradigms used for building software systems. However, despite its industrial and academic popularity, OOP is widely criticized for its high complexity, low maintainability, and a lack of agreed-upon and rigorous principles. Eolang (a.k.a. EO) was created to solve the said problems by restricting language features and introducing a formal object calculus and a programming language with a reduced set of features. In this initial phase of the research and development of Eolang programming language, we analyse the Eolang language, compare it to other OOP languages and describe the core features of this new language.

Keywords: EO. Elegant Objects. OOP paradigm. Functional programming. JVM. True OOP.

1 Problem

The fundamental problem in OOP is the lack of a rigorous formal model, the high complexity, the too many ad hoc designs, and the fact that programmers are not satisfied. Many OOP languages, including Java, have critical design flaws. Due to these fundamental issues, many Java-based software products are low quality and hard to maintain. The above drawbacks often result in system failures, customer complaints, and lost profits. The problem has been recognized; however, it has not been addressed yet.

In addition, OOP styles were considered to ensure the formation of effective compositions of software products. Among the formal approaches are the work of the theoretical models describing the theoretical OOPs. These include, for example, Abadi's work on sigma-calculus, which can be used to reduce the semantics of any object-oriented programming language to four elements: objects, methods, fields, and types. Further development of these works led to the creation of a phi-calculus used in the description of elementary design patterns.

1.1 The aim

This work aims to analyze the Eolang language, describe the syntax and semantics of Eolang to assess the quantitative and qualitative characteristics of Eolang through the prism of comparing Eolang with other object-oriented programming languages such as Java and C ++. Summarize and check compellability and utility in simple cases, making the solution available to the world developers' community.

2 Related Works

Kotlin language specification, Akhin [1], Comparative Studies of Programming Languages, Paquet [2] , Direct Interpretation of Functional Programs for Debugging, Whitington [3].

3 Solution

We want to stay as close to Java and JVM as possible, mostly in order to re-use the eco-system and libraries already available. We also want to have an ability to compile it to any other language, like Python, C/C++, Ruby, C-sharp, etc. In other words, EO must be platform independent.

3.1 Metrics

Before providing test cases we have chosen main metrics according to the current level of EO-compiler and EO-objects at all.

3.1.1 Syntax comparison. We decided to choose this metric to formalize and distinguish Emprical comparison. Rules written in RBNF (like a text in a programming language) consist of separate elements - tokens. Lexemes are the names of concepts that are called nonterminal symbols or simply nonterminals in the theory of formal languages. For example, in the rule StatementSequence = Operator ";" Operator. Nonterminals are OrderOperator and operator . Terminal symbols are the characters that make up the final (terminal - final, final) program. When writing to RBNF, terminal characters are written in quotation marks. In the given example, one terminal is ";". Terminal symbols are also RBNF tokens. Finally, tokens include special characters used in the RBNF itself. In the sequence of statements rule, these are the equal sign, curly braces, and a period at the end.

The total number of tokens in a language syntax description can serve as a generalized characteristic of the size of this description. It is much better to use the number of tokens as a measure of volume than, say, the number of characters in the description. In this case, the value of our criterion will not depend on what language (Russian, English) or what specific words the nonterminals are called - the concept of language. The number of different non - terminals is the next characteristic that we will calculate. The number of concepts used to describe a language is undoubtedly the most important property on which the ease of mastering this language depends. It can be noted that the number of nonterminals must be equal to the number of rules in the syntax description, since exactly one rule must exist for each concept. Set and the number of different terminal symbols of the language mentioned in the syntactic formulas characterize the vocabulary of the language - a set of characters and special characters. In all the languages we are discussing(EO,Java,Groovy), there are service words that can be used only in a strictly defined case. The programmer should actually know them by heart. Counting the number of service words will allow you to estimate the volume of cramming.

3.1.2 Empirical comparison based on OO constructs. LOC(lines of code), Readability, Maintainability, Cohesion and coupling,Code smells.

The work is : 1) To form a class-diagram for EO solution(optional). 2) To implement the same program with another language(Java,Groovy,Kotlin): 1 variant is to write an equivalent of EO solution. 2 variant is to write the program using non-efficient from the point of EO constructions(inheritance, type casting, flow control statements, etc.)

3.1.3 Execution Time and Memory Usage comparison. Measure the efficiency of Eolang, despite the fact of inefficient compiling process. We implement some prototypes to identify it. This prototypes are:

Pi digits
Binary tree
3 types of sorting algorithms.

They are: Bubble sort, Insertion sort, Selection sort.

3.1.4 Software reliability. During our comparison let's try to find out software reliability of solutions that use non-reliable(from EO postulat) construction(if these cases will be produced) and analyze cases in which this problem arises(if they will be.…)

The argument for this is that the only provement of non-reliable construction is only one that this is a procedural approach .

4 Discussion

We faced some issues connected to the bugs and optimization problems.Especially when we tried to implement new test cases, we were needed to implement new functionality and fix bugs that are caused by new implementations. All results are provided by weekly reports and github issue section https://github.com/cqfn/eo/issues.

5 Deliverables

5.1 Languages

As it was already mentioned we tried to be as close to JVM languages as possible We examine next languages:

5.1.1 Java. The Java programming language originated as part of a research project to develop advanced software for a wide variety of network devices and embedded systems. The goal was to develop a small, reliable, portable, distributed, real-time operating platform. When the project started, C++ was the language of choice. But over time the difficulties encountered with C++ grew to the point where the problems could best be addressed by creating an entirely new language platform. Design and architecture decisions drew from a variety of languages such as Eiffel, SmallTalk, Objective C, and Cedar/Mesa. The result is a language platform that has proven ideal for developing secure, distributed, network-based end-user applications in environments ranging from network-embedded devices to the World-Wide Web and the desktop.

5.1.2 Kotlin. Kotlin was designed to run on the JVM. It comes with its own compiler that compiles kotlin code to bytecode that can run on the JVM. The bytecode generated by kotlin compiler is equivalent to the bytecode generated by the Java compiler. Whenever two-byte code files run on JVM, due to their equivalent nature they can communicate with each other and that's how interoperability is established in kotlin for Java . Kotlin was developed keeping interoperability in mind. A Kotlin class or a function can refer to the Java classes and their methods in a simple way. Kotlin program files (.kt) and Java program files (.java) can be in the same project. They are all compiled and converted to .class files which are bytecodes.

5.1.3 Groovy. Groovy is a dynamic and optionally typed object oriented scripting language. Just like Kotlin and Scala, it is also interoperable well with Java; almost all Java code is also valid Groovy code.

5.2 Abstraction

5.2.1 EO. Already exist. The object abstracts behavior (or cohesion) of other objects LOC [x a b c] > polynomial (((x.pow 2).mul a).add (x.mul b)).add c > @

5.2.2 Java. To create an abstract class, just use the abstract keyword before the class keyword, in the class declaration. LOC class Polynomial public Polynomial(int x, int a, int b, int c) this.x=x; this.a=a; this.b=b; this.c=c; public int compute() return x*x*a + x*b + c;

5.2.3 Groovy. To create an abstract class, just use the abstract keyword before the class keyword, in the class declaration. LOC class Polynomial public Polynomial(int x, int a, int b, int c) this.x=x; this.a=a; this.b=b; this.c=c; public int compute() return x*x*a + x*b + c;

5.2.4 Kotlin. Like Java, abstract keyword is used to declare abstract classes in Kotlin. An abstract class cannot be instantiated (you cannot create objects of an abstract class). However, you can inherit subclasses from can them.

5.3 Encapsulation

5.3.1 EO. Doesn't exixt and will not be introduced

5.3.2 Java. Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

5.3.3 Groovy. In Groovy, everything is public. There's no idea of private fields or methods. At least, not in the way they would be represented in C++ or Java.

5.3.4 Kotlin. OOP encapsulation in Kotlin unlike Python is enforced and has some fine grained levels (scope modifiers/keywords)

5.4 Inheritance

5.4.1 EO. Doesn't exist and won't be introduced. The usual inheritance is presented by decorators(@) The main idea is that in the production(inheritance couses many problems) https://www.yegor256.com/2016/09/13/inheritance-is-procedural.html

5.4.2 Java. In Java, it is possible to inherit attributes and methods from one class to another. We group the ""inheritance concept"" into two categories: subclass (child) - the class that inherits from another class. superclass (parent) - the class being inherited from to inherit from a class, use the extends keyword.

5.4.3 Groovy. Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class). extends is the keyword used to inherit the properties of a class.

5.4.4 Kotlin. All classes in Kotlin have a common superclass Any, that is the default superclass for a class with no supertypes declared. Any has three methods: equals(), hashCode() and toString(). Thus, they are defined for all Kotlin classes.

By default, Kotlin classes are final: they can't be inherited. To make a class inheritable, mark it with the open keyword.

5.5 Polymorphism

5.5.1 EO. Will be implemented (Ad hoc polymorphism)

Duck typing in computer programming is an application of the duck test — "If it walks like a duck and it quacks like a duck, then it must be a duck " — to determine whether an object can be used for a particular purpose. With normal typing, suitability is determined by an object's type. In duck typing, an object's suitability is determined by the presence of certain methods and properties, rather than the type of the object itself.

5.5.2 Java. Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways. Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

5.5.3 Groovy. If you do this in Java (splitting the classes out to their own files of course), it won't compile. Java looks at the reference type for available methods, so you will get a NoSuchMethodException. In Groovy, however, it looks at the type of the object, not the type of the reference so the method is found at runtime.

5.5.4 Kotlin. Kotlin supports two forms of polymorphism because it is both strongly and statically typed. The first form of polymorphism happens when the code is compiled. The other form happens at runtime. Understanding both forms of polymorphism is critical when writing code in Kotlin.

5.6 Data types

5.6.1 EO. Presented like Atom Data Type https://stackoverflow.com/questions/10525511/what-is-the-atom-data-type

it is an acronym of "Access to Memory" or something like that. It is a term used for simple numerical identifiers (other name is ""handles"") which represent some internal data structures in the system. http://www.sharelatex.com

5.6.2 Java. Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double. Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

5.6.3 Groovy. Groovy supports the same number of primitive types as Java.

5.6.4 Kotlin. In Kotlin, everything is an object in the sense that we can call member functions and properties on any variable. Some of the types can have a special internal representation - for example, numbers, characters and booleans can be represented as primitive values at runtime - but to the user they look like ordinary classes.

5.7 Perfomance results

According to the provided tests we underline that the results are under pressure, because of lack of optimization. Especially we faced up with a problem of StackOverflowError while trying to compile hard tests(100,1000..etc. args).

5.7.1 Factorial of 6. [n] > factorial
if. > @
n.eq 0
1
n.mul (factorial (n.sub 1))
Program's consumption at runtime: 236 MB

5.7.2 Maximum of an array of 8 nums. [args...] > max
[accumulator current] > reduceFunction
if. > @
current.toInt.less accumulator
accumulator
current.toInt
reduce. > biggest
args
0
reduceFunction
stdout > @
sprintf
"
biggest
Program's consumption at runtime: 250 MB

5.7.3 Linear search for an array of 5 numbers. The last argument passed in the command line is the search key and the elements before it makes the array. .
[args...] > max
memory > key
memory > temp
(args.get 5).toInt > elem
memory > x
seq > @
key.write -1
x.write 0
while.
x.less 5
[i]
seq > @
temp.write (args.get i).toInt
(elem.eq temp).if
key.write i
false
x.write (x.add 1)
stdout
sprintf "Element is found at index
Program's consumption at runtime: 193 MB

6 Conclusion

While in the development of complex software systems, universal languages that support the multi-paradigm style are often used, there are extensive classes of problems that are well suited to object-oriented OO programming languages with increased reliability. As part of the work, the ways of creating such a language are considered. It has been shown that several unreliable designs are often unnecessary. Some of the presented designs can be replaced with more reliable ones.

Theoretically promising direction of development of languages with objects is a functional approach, in which the function acts as an object, and its arguments - as attributes and/or methods. In this case, it is possible to provide some neutrality both in terms of typing and in terms of function parameters representing attributes and methods. Additional benefits are associated with parametric polymorphism, type inference and computational strategies - call-by-name, call-by-value, call-by-need, etc. To a certain extent these mechanisms are implemented in the languages LISP,(S)ML, Miranda, Haskell. The multi-paradigm approach takes place in the programming language of the F-sharp, It is possible to "immerse" fragments of the source code written in a wide range of languages into an abstract/virtual machine (such as CLR or JVM) that broadcasts components into intermediate code (MSIL or byte) respectively. As we have

described above, existing OOP languages exhibit high complexities and many design flaws.

References

[1] Akhin. 2020. *Kotlin language specification.*
[2] Paquet. 2010. *Comparative Studies of Programming Languages.*
[3] Whitington. 2019. *Direct Interpretation of Functional Programs for Debugging.*