Towards A new Object-Oriented Programming language: Analysis of the Eolang Language and Compiler

Abstract – Object-Oriented Programming (OOP) has been the most popular programming paradigm employed in developing many programming languages over the years. This fame is partly because of the tremendous merits the paradigm exposes and the effectiveness in problem-solving. In spite of these merits and popularity, OOP languages exhibit issues that are inherent in applying the fundamental principles of the OOP paradigm that are widely criticized for many reasons, such as the lack of agreed-upon and rigorous principles. The Eolang programming language is a novel initiative that aims to ensure the proper practical application of the OOP paradigm. Pure objects, free from incorrectly made design decisions common for mainstream technologies, is the eponymous philosophy of Eolang. The purpose of this work is to analyse the current implementation of the Eolang language and compiler, point out the issues, and make suggestions for improvements. The main task is to analysis and assess the reliability of the proposed language solutions. This will include some qualitative and performance assessments of the compiler, its code generation and code execution time as well as comparison with Java programming language.

Introduction

Object Oriented Programming (OOP) has been the dominant paradigm in the software development industry over the past decades. OOP languages, including well-known languages such as Java, C++, C# and Python, are widely used by major technology companies, software developers, and leading providers of digital products and solutions for various projects [1]. It should be noted that virtually all key programming languages are essentially focused on supporting multiparadigm style, which allows for different style of coding in a single software project. The absence of restrictions on programming style often leads to the use of not the most reliable coding techniques, which greatly affects the reliability of programs in several areas. The existing attempts to limit the programming style, by directives, do not always lead to the desired result. In addition, supporting different programming paradigms complicates languages and tools, reducing their reliability. Moreover, the versatility of these tools is not always required everywhere. Often many programs can be developed using only the OOP paradigm [2].

Furthermore, among language designs considered OOP, there are those that reduce the reliability of the code being developed. Therefore, the actual problem is the development of such OOP languages that provide higher reliability of programs. This is especially true for a number of critical areas of their application.

A lot of teams and companies that use these languages suffer from the lack of quality of their projects despite the tremendous effort and resources that have been invested in their development. Many discussions concerning code quality issues appeared in the field. Mainly these focused on eliminating code smells and introducing best practices and design patterns into the process of software development. As many industry experts point out, the reason for project quality and maintainability issues might be explained by the essence of inherent flaws in the design of the programming language and the OOP paradigm itself, and not the incompetence or lack of proper care and attention of the developers involved in the coding solely [3]. Thus, it is necessary to develop new programming languages and approaches for implementing solutions in the OOP paradigm are to be developed. Some programming languages emerged based on the Java Virtual Machine to address this claim and solve the design weaknesses of Java for the sake of better quality

of produced solutions based on them. These are Groovy, Scala, and Kotlin, to name a few [4]. While many ideas these languages proposed were widely adopted by the community of developers, which led to their incorporation into the mainstream languages, some were considered rather impractical and idealistic. Nevertheless, such enthusiastic initiatives drive the whole OOP community towards better and simpler coding.

EO (stands for Elegant Objects or ISO 639-1 code of Esperanto) is an object-oriented programming language. It is still a prototype and is the future of OOP [5]. EO is one of the promising technologies that arise to drive the course of elaboration of the proper practical application of the OOP paradigm. The EO philosophy advocates the concept of so-called Elegant Objects, which is the vision of pure that is free from the incorrectly taken design decisions common for the mainstream technologies. Specifically, these are: Static methods and attributes; Classes; Implementation inheritance; Mutable objects; Null references; Global variables and methods; Reflection and annotations; Typecasting; Scalar data types; Flow control operators (for loop, while loop, etc.).

Eolang is an object-oriented programming language aimed at realizing the pure concept of object-oriented programming, in which all components of a program are objects. Eolang's main goal is to prove that fully object-oriented programming is possible not only in books and abstract examples but also in real program code aimed at solving practical problems.

Research 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 ς-calculus (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 ρ-calculus used in the description of elementary design patterns [2].

This work aims to provide an overview of Eolang, check the capability and functionality of the main aspects, assess and understand the features through the prism of comparing Eolang with other OOP languages (such as Java, Groovy, and Kotlin), with examples for simple use-cases and publish the R&D results. The comparative analysis will focus on, between these languages, the OOP principles, data types, operators and expressions, as well as declarative and executable statements.

Work in progress...

Analysis of the Eolang semantics features

The main task of this analysis is to assess the reliability of the proposed language solutions. We also analyse approaches to building a code generator that provides an acceptable efficiency of transformation of language constructions. In addition, an attempt is made to propose alternative solutions at the level of the programming language, which will improve the reliability of the development process, as well as a more efficient implementation of the compiler.

Abstract Object

*An object is **abstract** if at least one of its attributes is free—isn't bound to any object. An object is **closed** otherwise.*

--Yegor Bugaenko

The term "Abstract Object" itself contains a semantic contradiction. On one hand, the object seems to exist, but on the other hand, there is uncertainty in it, which in many cases does not allow working with it normally. That is, using such an object, you can try to access the data that has not yet been initialized. This leads to the interruption of the program during execution after a successful compilation. Many of these errors are difficult to detect, because when accessing an object using methods, it is not obvious whether it contains undefined attributes or they are absent in this method. The situation may be aggravated by the fact that in the language it is possible to produce new abstract objects with partial definition from other abstract objects.

This is where the specificity of the uncertainty of an abstract object manifests itself. Why create such objects if their direct use leads to additional errors?

Maybe instead it is worth talking about a class that describes the general construction of an object? From this point of view, it is impossible to create an object initially if there are inaccessible objects in it.

In a manner of speaking, if an attribute refers to an undefined object, then what is it? In particular, why is such an object needed if a whole series of internal methods (in the traditional sense) cannot work due to the absence of this attribute.

Example 1. Demonstration that abstract objects reduce the reliability of programming

File with rectangle:

```
1. +package rectangle2
2. +alias stdout org.eolang.io.stdout
3. +alias sprintf org.eolang.txt.sprintf
4.
5. [a b] > rectangle
6.
7.   sprintf > toString
8.     "It is Rectangle: a = %d, b = %d"
9.     a
10.      b
11.
12.    mul. > square
13.      a
```

Work in progress...

```
14.      b
15.
16.    mul. > perimeter
17.      add.
18.        a
19.        b
20.      2
21.
22.    stdout > out
23.      sprintf
24.        "It is Rectangle: a = %d, b = %d\n"
25.        a
26.        b
27.
28.    [r] > cmpSquare
29.      if. > @
30.        eq.
31.          r.square
32.          ^.square
33.        "Yes"
34.        "No"
35.
```

File with options for using a rectangle:

```
1. +package rectangle2
2. +alias rectangle rectangle2.rectangle
3. +alias stdout org.eolang.io.stdout
4. +alias sprintf org.eolang.txt.sprintf
5.
6. [args...] > app
7.
8.   rectangle > r
9.     (args.get 0).toInt
10.      (args.get 1).toInt
11.
12.    rectangle > r2
13.      3
14.      4
15.
16.    rectangle > r3
17.      13:b
18.      42:a
19.
20.    rectangle > r4
21.      13:b
22.
23.    rectangle > r5
24.      1313
25.
26.    rectangle > r6
27.
28.    r3 > r7
29.
30.    r7 > r8
```

```
31.      555
32.
33.    r8 > r9
34.      101010:b
35.
36.    seq > @
37.
38.      r.out
39.
40.      r.square > s!
41.      stdout
42.        sprintf
43.          "Square = %d\n"
44.          s
45.
46.      r.perimeter > p!
47.      stdout
48.        sprintf
49.          "Perimeter = %d\n"
50.          r.perimeter
51.
52.      stdout
53.        sprintf
54.          "r2: a = %d, b = %d\n"
55.          r2.a
56.          r2.b
57.      r2.out
58.
59.      stdout
60.        sprintf
61.          "r3: a = %d, b = %d\n"
62.          r3.a
63.          r3.b
64.      r3.out
65.
66.      stdout
67.        sprintf
68.          "r4: b = %d\n"
69.          r4.b
70.
71.      stdout
72.        sprintf
73.          "r5: a = %d\n"
74.          r5.a
75.
76.      stdout
77.        sprintf
78.          "r6: a = ???\n"
79.
80.      stdout
81.        sprintf
82.          "r7 as clone r3: a = %d, b = %d\n"
83.          r7.a
84.          r7.b
85.      r7.out
86.
87.      stdout
```

Work in progress...

```
88.         sprintf
89.           "r8 as clone r7: a = %d, b = %d\n"
90.           r8.a
91.           r8.b
92.       r8.out
93.
94.       stdout
95.         sprintf
96.           "r9 as clone r8: a = %d, b = %d\n"
97.           r9.a
98.           r9.b
99.       r9.out
```

Output:

```
It is Rectangle: a = 3, b = 5
Square = 15
Perimeter = 16
r2: a = 3, b = 4
It is Rectangle: a = 3, b = 4
r3: a = 42, b = 13
It is Rectangle: a = 42, b = 13
r4: b = 13
r5: a = 1313
r6: a = ???
r7 as clone r3: a = 42, b = 13
It is Rectangle: a = 42, b = 13
r8 as clone r7: a = 555, b = 13
It is Rectangle: a = 555, b = 13
r9 as clone r8: a = 555, b = 101010
It is Rectangle: a = 555, b = 101010
```

The example demonstrates that the use of partially defined (abstract) objects violates the integrity of their perception and functioning. Using object methods as well as undefined attributes leads to runtime errors. Only partial direct access to initialize attributes is possible. Almost all methods directly related to the processing of the state of the object cease to work.

Recommendation

In my opinion, it is necessary to prohibit the use of abstract objects in the executable code, since their use reduces the reliability of the program. However, by themselves, abstractions that are undefined in meaning can be useful. Therefore, one can simply return to the concept of a class. In principle, I do not see anything bad with a few variants of class constructors that provide partial initialization, although here, too, partial initialization can lead to a decrease in programming reliability. Also, nothing prevents keeping the cloning of objects at the level of language support, thereby eliminating the need to implement the Prototype pattern.

Work in progress...

Type system analysis

One of the features of the language is the lack of typing. That is, if we consider the original description, then the type system (both static and dynamic) is absent. Achieving the uniqueness of language constructions is determined mainly at the level of operational unambiguity, when the semantics of the operations performed is revealed as a combination of data objects and method objects. That is, there is a sense of analogy with assembly languages, in which the location of data in memory is not tied to the type, and the semantics of command execution is determined mainly by the operation code and, in some cases, by additional data identification through registers and memory access.

It should be noted that additional identification appeared in later assemblers, such as x86 assembler. Before that, many assemblers got by with identification through an opcode (for example, in pdp-11)

As a result, the use of implicit type conversions leads to a programming style that is often reminiscent of assembly. Before performing one or another operation defined by an object, you have to explicitly convert data to the desired object (equivalent to type conversion) or determine the semantics of operations using a constant (constant object) used as an argument.

Therefore, the type of object being processed is often impossible to predict. This is especially true when passing parameters. It is easy to generate errors related to the fact that instead of an object of one type, an object of a different type is passed for processing. The transformations of the transmitted object can be hidden by their location in other files or in some other way. You have to use an explicit cast at the point of object processing. However, there are problems associated with the fact that not all types of objects can be explicitly transformed. That is, in fact, we have a typeless representation of objects when passing parameters

Example 2. Demonstration of what the lack of typing leads to.

File:

```
1. +package book2
2. +alias stdout org.eolang.io.stdout
3. +alias sprintf org.eolang.txt.sprintf
4.
5. [args...] > app
6.   seq > @
7.     stdout "Hello, World!\n"
8.     transform 5
9.
10.      [isbn] > book2
11.        "Object Thinking" > title
12.        memory > price
13.        seq > @
14.          stdout
15.            sprintf "title: %s\n"
16.              title
```

```
17.          price.write 10.0
18.          stdout
19.            sprintf "price = %f\n"
20.              price
21.          price.write 200
22.          stdout
23.            sprintf "price = %d\n"
24.              price
25.          price.write "Hello, memory!"
26.          stdout
27.            sprintf "price = %s\n"
28.              price
29.
30.      book2 "cool book" > bbb
31.      stdout
32.        sprintf "bbb isbn = %s\n"
33.          bbb.isbn
34.      stdout
35.        sprintf "title = %s\n"
36.          bbb.title
37.      book2.price.write 1000
38.      stdout
39.        sprintf "book 2 price = %d\n"
40.          book2.price
41.      stdout
42.        sprintf "bbb price = %s\n"
43.          bbb.price
44.
45.      book2 666 > xxx
46.      stdout
47.        sprintf "xxx isbn = %d\n"
48.          xxx.isbn
49.
50.      book2 2.71828 > yyy
51.      stdout
52.        sprintf "yyy isbn = %f\n"
53.          yyy.isbn
54.
55.  [a] > transform
56.    stdout > @
57.      sprintf "%d\n" a
```

Output:

```
Hello, World!
5
title: Object Thinking
price = 10,000000
price = 200
price = Hello, memory!
title: Object Thinking
price = 10,000000
price = 200
price = Hello, memory!
bbb isbn = cool book
title = Object Thinking
```

```
book 2 price = 1000
bbb price = Hello, memory!
title: Object Thinking
price = 10,000000
price = 200
price = Hello, memory!
xxx isbn = 666
title: Object Thinking
price = 10,000000
price = 200
price = Hello, memory!
yyy isbn = 2,718280
```

The example shows that a typeless solution leads to the dualism of using the same attributes. This leads to the fact that even generated objects become incomparable with each other. Especially when the connection between them is not tracked in the future. It should also be noted that cloning leads to copying the internal states of objects, which is not always advisable.

Identity (originating from one) of objects can be modeled using a common constant that identifies the type of the object. But this is again a purely assembler trick that does not allow object matching at compile time.

Recommendation
Introduce a static type system into the programming language, similar to modern OO programming languages, with possible enhancements to improve data control at compile time. As an addition at subsequent stages of the language development for methods with static polymorphism (method name overloading), you can introduce type parsing and type inference, characteristic of pure functional programming languages. Perhaps in some kind of truncated version.

The introduction of static typing will improve the reliability of programs and provide additional control both during compilation and when using static code analyzers.

Processing Alternatives
The handling of alternative objects that have similar behavior is one of the common situations in dynamic linking. In the procedural approach, alternatives are usually analyzed explicitly either by checking the attached type (explicit indication of the type of the alternative is used, for example, in the union of the Ada programming language), or by using a sign (key) explicitly entered by the programmer (for example, in the C language), which simulates the variant notation programming language Pascal.

In the case of the OO approach, explicit type checking in most cases is replaced by "duck typing" in which any object that executes some predefined method, also existing in other objects, can execute its code, which is determined by the internal implementation of this method. In this case, OO has polymorphism, which in most modern OO languages is implemented through the inheritance (or extension) mechanism, when the methods of the base class are overridden in derived classes. This approach is usually implemented through virtual method tables, which provide reasonably fast access to virtual methods.

Work in progress...

Another option for implementing polymorphism is to use associative access (associative dynamic polymorphism), when an object containing some methods, having received a message from another object, looks for a method in its method table. Having matched the received message by a model and having found the appropriate method, it carries out its execution. If absent, the exception is handled. This approach is much slower than the previous one. However, it allows, through the identity of the methods, to bind and launch objects that are not otherwise related to each other.

It can also be noted that support for polymorphism is currently possible in procedural languages as well. However, it has been implemented using different principles. In particular, Go duck typing is implemented through the introduction of interfaces. A similar mechanism is used in the Rust programming language. Also, in a number of works (mine), procedural-parametric polymorphism was proposed and experimentally demonstrated on the extension of the Oberon-2 language.

It is also worth noting that all types of polymorphism, in principle, can be implemented in functional programming languages.

EOLANG uses associative dynamic polymorphism, where unrelated objects can execute their methods with the same signature, connecting to some object. This connection can be mutable or random. Then it is not known which of the objects is connected at the moment. And only by its visible behavior (and it may not always be visible) it is possible to determine which of the objects is currently connected.

File with polymorphism:

```
1. +package fig01
2. +alias rectangle fig01.rectangle
3. +alias triangle fig01.triangle
4. +alias stdout org.eolang.io.stdout
5. +alias sprintf org.eolang.txt.sprintf
6.
7.  [f] > figure
8.   stdout > out
9.     sprintf "Figure. %s\n"
10.        f.toString
11.
12.   [args...] > app
13.
14.    rectangle 10 20 > r
15.    triangle 3 4 5 > t
16.    figure r > figR
17.    figure t > figT
18.    memory > m
19.
20.    seq > @
21.
22.      r.out
23.      r.perimeter > pr!
24.      stdout
25.        sprintf
26.          "Perimeter = %d\n"
27.          pr
```

Work in progress...

```
28.
29.      t.out
30.      t.perimeter > pt!
31.      stdout
32.        sprintf
33.          "Perimeter = %d\n"
34.          pt
35.
36.      figR.out
37.      figT.out
```

File with triangle:

```
1. +package fig01
2. +alias stdout org.eolang.io.stdout
3. +alias sprintf org.eolang.txt.sprintf
4.
5. [a b c] > triangle
6.   2 > @
7.   add. > perimeter
8.     add.
9.       a
10.        b
11.      c
12.
13.    stdout > out
14.      sprintf
15.        "It is Triangle: a = %d, b = %d, c = %d\n"
16.        a
17.        b
18.        c
19.
20.    sprintf > toString
21.      "It is Triangle: a = %d, b = %d, c = %d"
22.      a
23.      b
24.      c
25.
26.    [t] > cmpSquare
27.      if. > @
28.        eq.
29.          t.square
30.          ^.square
31.        "Yes"
32.        "No"
```

File with rectangle:

```
1. +package fig01
2. +alias stdout org.eolang.io.stdout
3. +alias sprintf org.eolang.txt.sprintf
4.
5. [a b] > rectangle
6.   1 > @
```

Work in progress...

```
7.   sprintf > toString
8.     "It is Rectangle: a = %d, b = %d"
9.     a
10.      b
11.
12.    mul. > perimeter
13.      add.
14.        a
15.        b
16.      2
17.
18.    stdout > out
19.      sprintf
20.        "It is Rectangle: a = %d, b = %d\n"
21.        a
22.        b
23.
24.    [r] > cmpSquare
25.      if. > @
26.        eq.
27.          r.square
28.          ^.square
29.        "Yes"
30.        "No"
```

The presence of duck typing cannot always be used to accurately identify an object. This is due to the fact that in some cases a preliminary type analysis is required before applying the methods. The lack of explicit type checking in the language makes it impossible to identify the connected object. However, this check can be implemented programmatically by mapping to each of the objects an appropriate constant or method that returns the type of the object. Typically, even statically typed languages use dynamic run-time type checking for objects derived from derived classes.

In a dynamically typed system, the type of an object is also implicitly specified. However, this type is stored inside an object and can be explicitly checked at runtime.

In order to explicitly check the type of an object, it is necessary to introduce an additional attribute into it, which is returned by the specified attribute. For example, an integer. Then the comparison of objects for similarity is carried out using the same value of this attribute.

Duck typing used in the language is based on the similarity of methods for processing objects, regardless of their relationship, for example, through the mechanism of inheritance. This allows you to connect a variety of objects, which leads to a decrease in the reliability of the program code. In addition, for such a connection, a mechanism based on associative arrays is usually used, which slows down the calling of methods (the call is carried out by name with the search for the desired name in the map).

It should also be noted that the typeless mechanism allows you to call methods with the same names, but used for completely different purposes, which also leads to a decrease in the reliability of the program.

Unfortunately, it was not possible to implement the example with memory. When I wanted to use polymorphism with the same mutable object, connecting sequentially different shapes to it and calling a polymorphic method. Apparently, in the current version, polymorphic work through memory is not implemented. Although in many programs there is just a substitution of one object for another.

Recommendation
The use of associative dynamic polymorphism (ADP) in the language allows flexible substitution of objects, using the call for different objects of the same message. However, it should be noted that this method is more suitable for dynamically typed languages. The use of a typeless organization of objects in the language (when it is impossible to explicitly and specifically indicate or determine the type) leads to a decrease in the reliability of the software being developed.

As a variant of the development of a language that uses the implemented approach to duck typing, we can propose the addition of dynamic typing of objects. Then, if necessary, you can explicitly check the type of the object without explicitly using the corresponding constants. However, in this case too, excessive flexibility in the organization of polymorphism can lead to a decrease in the reliability of the code and its control due to the substitution of any object whose method signature allows you to do this. On the other hand, ADP is slow enough to be used in real-time systems.

More reliable, in my opinion, is the use of static typing in combination with methods for supporting polymorphism oriented towards this, which allows to control the range of connected objects. It is also advisable to use a dynamic type checking mechanism at runtime, which is usually implemented in almost all OO languages.

Analysis of Code Generation and execution time
To Do

Conclusion and Recommendation
To Do

References
To Do