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. We show that the proposed model is less verbose and has a better execution time.

Introduction

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 less reliable coding techniques, which greatly affects the reliability of programs in several areas. The existing attempts to limit the programming style, by coding standards, 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. 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 on the proper practical application of the OOP paradigm. The EO philosophy advocates the concept of so-called Elegant Objects, thus, pure objects free from the incorrectly taken design decisions common to 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 EO programming language is a research and development project that remains in an undeveloped state. Eolang emerged from the “Elegant Objects” philosophy that advocates the vision of pure OOP programming, free from incorrectly made design decisions common for the mainstream technologies. The language is designed to curb the issues related to static code entities, inheritance, classes, mutable objects, null references, reflection, and global variables, to name a few.

Code Generation and Execution Time

The current implementation of the Eolang **translator (compiler),** which is hosted on GitHub, generates cumbersome Java source program that includes several lines of redundant codes. In addition, the execution time of the code is a little longer than manageable.

For example, the Eolang program below sums array elements and outputs to the console.

File app.eo

```
1. +package sandbox
2. +alias stdout org.eolang.io.stdout
3. +alias sprintf org.eolang.txt.sprintf
4.
5. [args...] > app
6.   memory > sum
7.   memory > count
```

Work in progress...

```
8.   seq > @
9.     count.write 0
10.      sum.write 0
11.      while.
12.        count.less ($.args.length)
13.        [i]
14.          seq > @
15.            ^.sum.write (^.sum.add (^.args.get i).toInt)
16.            ^.count.write (^.count.add 1)
17.      stdout
18.        sprintf "Sum is %d\n" sum
```

The Java source code generated by the compiler is shown below:

File EOapp.java

```
1.  package sandbox;
2.
3.  import org.eolang.*;
4.  import org.eolang.phi.*;
5.
6.  public final class EOapp extends PhDefault {
7.      public EOapp() {
8.          this(new PhEta());
9.      }
10.     public EOapp(final Phi parent) {
11.         super(parent);
12.         this.add("args", new AtVararg(/* default */));
13.         this.add("sum", new AtBound(new AtOnce(new AtLambda(this, self -> {
14.             Phi ret = new org.eolang.EOmemory(self);
15.             return ret;
16.             })));
17.         this.add("count", new AtBound(new AtOnce(new AtLambda(this, self -> {
18.             Phi ret = new org.eolang.EOmemory(self);
19.             return ret;
20.             })));
21.         this.add("φ", new AtBound(new AtOnce(new AtLambda(this, self -> {
22.             Phi ret = new org.eolang.EOseq(self);
23.             ret = new PhCopy(ret);
24.             Phi ret_1_base = new PhMethod(self, "count");
25.             Phi ret_1 = new PhMethod(ret_1_base, "write"); // why instantiate
26.             ret_1 = new PhCopy(ret_1); // make a copy right after
27.             Phi ret_1_1 = new org.eolang.EOint(self);
28.             ret_1_1 = new PhCopy(ret_1_1);
29.             ret_1_1 = new PhWith(ret_1_1, "Δ", new Data.Value<Long>(0L));
30.             ret_1 = new PhWith(ret_1, 0, ret_1_1);
31.             Phi ret_2_base = new PhMethod(self, "sum");
32.             Phi ret_2 = new PhMethod(ret_2_base, "write"); //why instantiate
33.             ret_2 = new PhCopy(ret_2); // and make a copy right after
34.             Phi ret_2_1 = new org.eolang.EOint(self);
35.             ret_2_1 = new PhCopy(ret_2_1);
36.             ret_2_1 = new PhWith(ret_2_1, "Δ", new Data.Value<Long>(0L));
37.             ret_2 = new PhWith(ret_2, 0, ret_2_1);
38.             Phi ret_3_base_base = new PhMethod(self, "count");
39.             Phi ret_3_base = new PhMethod(ret_3_base_base, "less"); // why instantiate
40.             ret_3_base = new PhCopy(ret_3_base); // and make a copy after
41.             Phi ret_3_base_1_base_base = self;
42.             Phi ret_3_base_1_base = new PhMethod(ret_3_base_1_base_base, "args");
43.             Phi ret_3_base_1 = new PhMethod(ret_3_base_1_base, "length");
44.             ret_3_base = new PhWith(ret_3_base, 0, ret_3_base_1);
45.             Phi ret_3 = new PhMethod(ret_3_base, "while");
46.             ret_3 = new PhCopy(ret_3); // Redundant
47.             Phi ret_3_1 = new EOapp$EO3$EO2$EOα1(self);
48.             ret_3_1 = new PhCopy(ret_3_1); // Redundant
49.             ret_3 = new PhWith(ret_3, 0, ret_3_1);
```

```
50.             Phi ret_4 = new org.eolang.io.EOstdout(self);
51.             ret_4 = new PhCopy(ret_4); // Redundant
52.             Phi ret_4_1 = new org.eolang.txt.EOsprintf(self);
53.             ret_4_1 = new PhCopy(ret_4_1); // Redundant
54.             Phi ret_4_1_1 = new org.eolang.EOstring(self);
55.             ret_4_1_1 = new PhCopy(ret_4_1_1); // Redundant
56.             ret_4_1_1 = new PhWith(ret_4_1_1, "Δ", new Data.Value<String>("Sum is
57.                     %d\n"));
58.             Phi ret_4_1_2 = new PhMethod(self, "sum");
59.             ret_4_1 = new PhWith(ret_4_1, 0, ret_4_1_1);
60.             ret_4_1 = new PhWith(ret_4_1, 1, ret_4_1_2);
61.             ret_4 = new PhWith(ret_4, 0, ret_4_1);
62.             ret = new PhWith(ret, 0, ret_1); // redundant
63.             ret = new PhWith(ret, 1, ret_2); // redundant
64.             ret = new PhWith(ret, 2, ret_3); // redundant
65.             ret = new PhWith(ret, 3, ret_4); // only one is relevant
66.             return ret;
67.             })));
68.         }
69. }
```

As we can see in the above generated code, an object is created (line 22), then after a copy of this object is immediately made (line 23), thus a copy of the object is assigned to a new variable. It is not clear why this is being done. This peculiarity exists throughout the code (There are several lines with “new PhCopy(***)” that logically seem redundant). Also, the codes on lines #62, 63 and 64 seem redundant. Thus the “new PhWith(***)” has been redundantly used on the some lines. After removing the blocks that generate the copying code from the compiler, the behavior of the compiled program did not change.

Now let us consider the Java program below which achieves the same objective, and compare the execution time:

```
1. class Main {
2.     public static void main(String[] args) {
3.         int numbers[] = {1, 2, 3, 4…}; // Array of 2700 elements
4.         System.out.println(getSum(numbers));
5.     }
6.
7.     public static int getSum(int[] elems){
8.         int sum = 0;
9.         int i=0;
10.          while(i<elems.length){
11.              sum += elems[i];
12.              i++;
13.          }
14.          return sum;
15.      }
16.
17.  }
```

Table 1 show the execution time comparison between Java program and the Eolang program

Table 1. Execution Time Comparison

	Java	**Eolang**
Input	Array of 2700 elements	Array of 2700 elements

Execution Time	0.2086576s	3.2857814s
Output	136350	136350

From the table we can see that it takes more time to execute the Eolang code on the mention test case as compared to the pure styled Java code.

Analysis of the Eolang semantics features (To Do)

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

TO DO

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.

Work in progress...

Contribution (aims and objectives)

To address these issues, we propose a new code generation for the Eolang runtime, an Eo-to-Java model, where we map Eolang object to Java class, a copy of an object is mapped to an instance of a class, Eolang free attributes map to class fields and a getData method for dataization. We then write programs with this model, assuming the Eolang style, and then compare to codes generated by the current Eolang runtime.

Proposed approach

We propose a new model for the basic runtime of the Eolang translator (compiler).

To Do

Eo To Java Transpiling Model

Table 3. Basic Runtime Classes

Class	Description
EOObject	Has four main objectives: 1. Instantiate an EO object (i.e. make a copy of it via a constructor) 2. Dataize an EO object 3. Set a parent object 4. Access attributes of an object
EOData	Used to store data primitives.
EODataObject	A wrapper class to interpret EOData as EOObject
EONoData	A primitive class representing the absence of data

Table 4. Mapping EO Entities to Java Code Structures

Eolang	Java
Abstraction (introduction of a conceptually new object)	A plain old Java class extending EOObject
Application (object copying)	Creating a new class instance
A free attribute	A class field that should be sat via the constructor
A bound attribute	A separate class with the following naming pattern EO<base>$EO<attr>
Duck typing	Everything is EOObject. Class fields are accessed via Java Reflection
Object dataization	Calling _getData()

Example Program(s) based on the proposed model

File EOfactorial.java (from current implementation)

```
1.  package sandbox;
2.
3. import org.eolang.*;
4. import org.eolang.phi.*;
5.
6. public final class EOfactorial extends PhDefault {
7.   public EOfactorial() {
8.     this(new PhEta());
9.   }
10.    public EOfactorial(final Phi parent) {
11.      super(parent);
12.      this.add("n", new AtFree(/* default */));
13.      this.add("φ", new AtBound(new AtOnce(new AtLambda(this, self -> {
14.        Phi ret_base_base = new PhMethod(self, "n");
15.        Phi ret_base = new PhMethod(ret_base_base, "eq");
16.        ret_base = new PhCopy(ret_base);
17.          Phi ret_base_1 = new org.eolang.EOint(self);
18.          ret_base_1 = new PhCopy(ret_base_1);
19.            ret_base_1 = new PhWith(ret_base_1, "Δ", new
   Data.Value<Long>(0L));
20.          ret_base = new PhWith(ret_base, 0, ret_base_1);
21.        Phi ret = new PhMethod(ret_base, "if");
22.        ret = new PhCopy(ret);
23.          Phi ret_1 = new org.eolang.EOint(self);
24.          ret_1 = new PhCopy(ret_1);
25.            ret_1 = new PhWith(ret_1, "Δ", new Data.Value<Long>(1L));
26.          Phi ret_2_base = new PhMethod(self, "n");
27.          Phi ret_2 = new PhMethod(ret_2_base, "mul");
28.          ret_2 = new PhCopy(ret_2);
29.            Phi ret_2_1 = new EOfactorial(self);
30.            ret_2_1 = new PhCopy(ret_2_1);
31.              Phi ret_2_1_1_base = new PhMethod(self, "n");
32.              Phi ret_2_1_1 = new PhMethod(ret_2_1_1_base, "sub");
33.              ret_2_1_1 = new PhCopy(ret_2_1_1);
34.                Phi ret_2_1_1_1 = new org.eolang.EOint(self);
35.                ret_2_1_1_1 = new PhCopy(ret_2_1_1_1);
36.                  ret_2_1_1_1 = new PhWith(ret_2_1_1_1, "Δ", new
   Data.Value<Long>(1L));
37.                ret_2_1_1 = new PhWith(ret_2_1_1, 0, ret_2_1_1_1);
38.              ret_2_1 = new PhWith(ret_2_1, 0, ret_2_1_1);
39.            ret_2 = new PhWith(ret_2, 0, ret_2_1);
40.          ret = new PhWith(ret, 0, ret_1);
41.          ret = new PhWith(ret, 1, ret_2);
42.        return ret;
43.      })));
44.    }
45.  }
```

File EOFactorial.java (from Proposed model)

```
1. package eo.test;
```

```
2.
3. import eo.org.eolang.calc.*;
4. import eo.org.eolang.core.*;
5. import eo.org.eolang.io.EOstdout;
6. import eo.org.eolang.txt.EOsprintf;
7.
8. public class EOFactorial extends EOObject {
9.     private EOObject n;
10.
11.      public EOFactorial(EOObject n){
12.          this.n = n._setParent(this);
13.      }
14.
15.      @Override
16.      public EOData _getData() {
17.          EOData res = new EOif(
18.                  new EOless(n, new EODataObject(2L)),
19.                  new EODataObject(1L),
20.                  new EOmul( _getAttribute("n"),
   _getAttribute("Factorial", new EOsub(_getAttribute("n"), new
   EODataObject(1L))))
21.
22.          )._setParent(this)._getData();
23.          return res;
24.      }
25.  }
26.
```

Runtime Performance comparison (current implementation vs proposed approach)
Factorial of 31

Table 2. Execution Time Comparison

Current Implementation	Proposed Model
0m1.179s	0m0.144s

Conclusion and Recommendation

To Do

References

To Do

Appendix

To Do