EOLANG Compiler Development Embedded into the Pipeline of the EO Maven Wrapper Plugin

Hadi Saleh
hsalekh@hse.ru
National Research University
Higher School of Economics
Moscow, Russia

Sergey Zykov
szykov@hse.ru
National Research University
Higher School of Economics
Moscow, Russia

Alexander Legalov
alegalov@hse.ru
National Research University
Higher School of Economics
Moscow, Russia

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 which 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 current Eolang compiler is verbose, less efficient compared to Java and shows limited potential for scalability. The purpose of this work is to analyse the current implementation of the Eolang compiler, identify the main issues, and propose a new solution for improvements. The main task is to develop a new Eolang compiler with a more efficient and reliable code generation and make it accessible at GitHub. The tasks will include some 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, has a better execution time and easier to maintain.

KEYWORDS

Elegant Objects, Eolang, XSLT, Compiler, Java, OOP, Maven, Decoration, Dataization, Duck typing

ACM Reference Format:
Hadi Saleh, Sergey Zykov, and Alexander Legalov. 2021. EOLANG Compiler Development Embedded into the Pipeline of the EO Maven Wrapper Plugin. In *Bari '15: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, October 11–15, 2021, Bari, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/1122445.1122456

ESEM 2021, October 11–15, 2021, Bary, Italy

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Object-Oriented Programming (OOP) is a prominent programming paradigm based on the concept of objects, yet the choice of this paradigm for designing programming languages has subtly lead to the design of languages that are not truly OOP, because of several reasons such a multiple trade-offs for achieving certain desirable properties [11]. Besides the trade-offs, the lack of rigorous formal model is fundamental problem [7] that paves way for several as-hoc designs. There is hardly a uniformity or an agreement on the set of features and mechanisms that belong in an OOP language and thus, making the paradigm itself far too general [12]. Khanam [10] affirms that OOP promotes ease in designing reusable software but the long coding methods makes it unreadable and enhances the complexity of the methods."

It should also 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]. Also, 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 [13]. This inspires the need 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 [14]. 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 [6]. 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.) [5, 6, 8].

1.1 Brief Description of The Current Eolang Compiler

The current Eolang compiler folder structure includes three main root directories:

(1) **eo-maven-plugin:** contains the transpiler and the maven wrapper plugin
(2) **eo-parser:** contains the lexer and the parser of EO source code. Produces XML documents that describe the input EO program.
(3) **eo-runtime** contains the classes of the EO programming language standard library (including the classes that simulate the EO language in the runtime).

The entire process of turning a ".eo" program into an executable binary code consists of a few steps, which must be done one after the other [https://github.com/cqfn/eo]. this is shown in Fig 1:

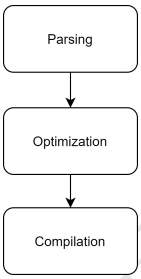

Figure 1: Transformation Process. Parsing is done first, optimization follows, and then, finally, compilation.

(1) **Parsing:** This is done by the org.eolang.parser.Syntax class in the eo-parser module. It takes the source code in a plain text format and parses into XML document, using ANTLR4 and Xembly. The output of the parser, after compilation, can be found in the target/eo/parse directory.
(2) **Optimization:** There are several XSL transformations that need to be done with the XML document to make it ready for compilation. Each transformation has its own .xsl file in the eo-parser directory. The class org.eolang.parser.Program is responsible for making XSLT transformations and the entire list of them is stored in the org.eolang.parser.Pack class. Some of XLST files are sanity checks (or linters). The output of each transformation you can find in the target/eo/optimize directory.
(3) **Compilation:** The class org.eolang.maven.CompileMojo in the eo-maven-plugin module is responsible for putting parsing and optimization steps together and then transforming the XML document into a collection of .java files. There are several transformations that do this, they all exist in .xsl files. The output of this step you can find in the target/generated-sources directory.

The eo-runtime module, which includes both .eo and .java codes, is the standard library for the most popular and important objects that will be needed to write Eolang programs. There are objects like string, int, sprintf, stdout, and to mention but a few.

1.2 Brief Description of Eolang with a sample Program

This is a program for a rectangle object that calculates area and perimeter. The documentation of the language is available at [8].

1.2.1 *File rectangle.eo.*

```
[a b] > rectangle
  a.mul b > area
  [] > perimeter
    2 > a
    mul. > @
      a
      add.
        ^.a
        ^.b
```

Description: In Eolang, everything is an object. Objects contains attributes which are also objects. From the simple program above:

(1) a is free attribute
(2) b is free attribute
(3) area is an application-style bound attribute
(4) perimeter is an abstraction-style bound attribute

2 RESEARCH PROBLEM

The EO programming language is a research and development project that remains in an undeveloped state. 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 [EO Yegor]. The current version of the compiler is synthetic and just a text translation of EO. The target code merely simulates the behaviour of objects in EO with poor performance and keeps many objects in memory. It also 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. Above all, it shows potential difficulty in scalability in case of future development. For instance, scalability could be a daunting task in the event of improving the type system. The problem of verbose and redundant code generation and the poor execution time is demonstrated in the following subsections.

2.1 Code Generation

The Eolang program below sums array elements and outputs to the console.

File app.eo

```
1. +package sandbox
```

```
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf

[args...] > app
  memory > sum
  memory > count
  seq > @
    count.write 0
    sum.write 0
    while.
      count.less ($.args.length)
      [i]
        seq > @
       ^.sum.write (^.sum.add (^.args.get i).toInt)
          ^.count.write (^.count.add 1)
    stdout
      sprintf "Sum is %d\n" sum
```

The Java source code generated by the compiler is shown below:
File EOapp.java

```
package sandbox;

import org.eolang.*;
import org.eolang.phi.*;

public final class EOapp extends PhDefault {
    public EOapp() {
        this(new PhEta());
    }
    public EOapp(final Phi parent) {
        super(parent);
        this.add("args", new AtVararg(/* default */));

        this.add("sum", new AtBound(new AtOnce(new AtLambda(this, self -> {
            Phi ret = new org.eolang.EOmemory(self);
            return ret;
            }))));
        this.add("count", new AtBound(new AtOnce(new AtLambda(this, self -> {
            Phi ret = new org.eolang.EOmemory(self);
            return ret;
            }))));
        this.add(" ", new AtBound(new AtOnce(new AtLambda(this, self -> {
            Phi ret = new org.eolang.EOseq(self);
            ret = new PhCopy(ret);
            Phi ret_1_base = new PhMethod(self, "count");
            Phi ret_1 = new PhMethod(ret_1_base, "write"); // why instantiate
            ret_1 = new PhCopy(ret_1); // make a copy right after
            Phi ret_1_1 = new org.eolang.EOint(self);
            ret_1_1 = new PhCopy(ret_1_1);
            ret_1_1 = new PhWith(ret_1_1, "", new Data.Value<Long>(0L));
            ret_1 = new PhWith(ret_1, 0, ret_1_1);
            Phi ret_2_base = new PhMethod(self, "sum");
            Phi ret_2 = new PhMethod(ret_2_base, "write"); //why instantiate
            ret_2 = new PhCopy(ret_2); // and make a copy right after
            Phi ret_2_1 = new org.eolang.EOint(self);
            ret_2_1 = new PhCopy(ret_2_1);
            ret_2_1 = new PhWith(ret_2_1, "", new Data.Value<Long>(0L));
            ret_2 = new PhWith(ret_2, 0, ret_2_1);
            Phi ret_3_base_base = new PhMethod(self, "count");
            Phi ret_3_base = new PhMethod(ret_3_base_base, "less"); // why instantiate
            ret_3_base = new PhCopy(ret_3_base); // and make a copy after
            Phi ret_3_base_1_base_base = self;
            Phi ret_3_base_1_base = new PhMethod(ret_3_base_1_base_base, "args");
            Phi ret_3_base_1 = new PhMethod(ret_3_base_1_base, "length");
            ret_3_base = new PhWith(ret_3_base, 0, ret_3_base_1);
            Phi ret_3 = new PhMethod(ret_3_base, "while");
            ret_3 = new PhCopy(ret_3); // Redundant
            Phi ret_3_1 = new EOapp\$EO3\$EO2\$EO\1(self);
            ret_3_1 = new PhCopy(ret_3_1); // Redundant
            ret_3 = new PhWith(ret_3, 0, ret_3_1);
            Phi ret_4 = new org.eolang.io.EOstdout(self);
            ret_4 = new PhCopy(ret_4); // Redundant
            Phi ret_4_1 = new org.eolang.txt.EOsprintf(self);
            ret_4_1 = new PhCopy(ret_4_1); // Redundant
            Phi ret_4_1_1 = new org.eolang.EOstring(self);
            ret_4_1_1 = new PhCopy(ret_4_1_1); // Redundant
            ret_4_1_1 = new PhWith(ret_4_1_1, "", new Data.Value<String>("Sum is
                    %d\n"));
            Phi ret_4_1_2 = new PhMethod(self, "sum");
            ret_4_1 = new PhWith(ret_4_1, 0, ret_4_1_1);
            ret_4_1 = new PhWith(ret_4_1, 1, ret_4_1_2);
            ret_4 = new PhWith(ret_4, 0, ret_4_1);
            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.    }
```

2.2 Execution time

Table 1 shows the execution time comparison between Java program and the Eolang program.

Table 1: Comparison of execution time

	Java	Eolang
Input	Array of 2700 elements	
Execution time	0.2086576s	3.2857814s
Output	136350	136350

Contribution: To address these issues, we propose a new Eo-to-Java translation 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. Then we implement this model and embed the new transpiler and runtime in the EO compiler pipeline. Also, we write programs with this model, assuming the Eolang style, and then compare to codes generated by the current Eolang transpiler.

3 PROPOSED SOLUTION

We propose a new model for the basic runtime and transpilation of the Eolang compiler without using XSLT as used in the current compiler implementation. The proposed solution will include two main development:

(1) Transpilation model, to be wrapped in a maven wrapper plugin
(2) Runtime model

Programming language used in the implementation of the source code of the translator and the runtime library is Java. The parser (as well as the lexer) is implemented (and augmented) as part of the ANTLR4 solution. The translation of the output code in Java is done through the construction and processing of an intermediate model obtained through parsing XML documents of the stage of parsing a program in EO. The final solution should be delivered in an easy-to-use form: a Maven plugin in the Maven Central repository.

3.1 Eolang Objects Translation

3.1.1 Naming and scoping. An Eolang object named "obj" of the global scope is translated into a Java class named "EOobj.java". The prefix "EO" is used as a simple measure to escape naming conflicts with Java code. Eolang objects of local scopes (bound attributes of other objects or anonymous objects passed as arguments during application) are not present as separate Java files. Instead, these are put to the scopes they originate from. So, attributes are named private inner classes, and anonymous EO objects are local Java classes. This technique makes target code denser and delegates the management of scopes and child-parent hierarchies to the Java Virtual Machine (JVM), not the Eolang runtime. The source program "obj" object is translated into a single "EOobj" class in a single EOobj.java file. No manual naming and class connections management. The transpiler is to compile the source program to the target platform as it is organized originally and let the Java Virtual Machine deal with all the relationships, hierarchies, and scopingAn Eolang object named "obj" of the global scope is translated into a Java class named "EOobj.java". The prefix "EO" is used as a simple measure to escape naming conflicts with Java code. Eolang objects of local scopes (bound attributes of other objects or anonymous objects passed as arguments during application) are not present as separate Java files. Instead, these are put to the scopes they originate from. So, attributes are named private inner classes, and anonymous objects are anonymous Java classes. This technique makes target code denser and delegates the management of scopes and child-parent hierarchies to the Java Virtual Machine (JVM), not the Eolang runtime. The source program "obj" object is translated into a single "EOobj" class in a single EOobj.java file. No manual naming and class connections management. The transpiler is to compile the source program to the target platform as it is organized originally and let the Java Virtual Machine deal with all the relationships, hierarchies, and scoping mechanisms locally (native).

3.1.2 Target Class Hierarchy. The target class extends org.eolang.core.EOObject. Constructors The target class has only one constructor for simplicity. There are two possible cases:

(1) If the source class has no free attributes, the resulting constructor has no arguments.
(2) If the source class has free attributes, the resulting constructor has all the arguments in the order specified in the source program. All arguments of the constructor are of type "EOObject", so the object does not know the actual type. In the case where free attributes are present, the default constructor (in Java) is disabled.

3.1.3 EO Objects Free Attributes Translation. Free attributes of the source object are translated as follows. Assuming there are these free attributes in the source Eolang object (in the order specified):

(1) a
(2) b
(3) c
(4) class
(5) he133

For each of them, the following will be performed:

(1) Generation of a private final field named EO<attributeName>: "EO" prefix is used to allow using some special names from the source language (e.g., class) that may be a reserve word in the target platform (Java). The field is of type EOObject.
(2) 2. Generation of a public method named EO<name>: The method's return type is EOObject. The method's body is just return this.EOattr;. 3. The body of the constructor sets the private EO<attributeName> field to the value of the argument EOattr. The proposed model is fully immutable since a code fragment that applies (or copies) a class can set the free attributes only once through the constructor. Fields holding free attributes are private and final. The default (empty) constructor is disabled. Once set, an attribute cannot be changed either from a user code fragment or from the inside of a class itself.Hence, objects are fully immutable.

However, the model has some weaknesses. First, it cannot handle the partial application mechanism. The proposed solution allows the "fully-applied-at-once" copying technique only. Secondly, just as in the original transpiler, the model does not perform type checking of the objects being passed for free attributes binding.

3.1.4 Eolang Objects Bound Attributes Translation. Bound attributes of the source object are translated as follows:

(1) For every bound attribute with an arbitrary name such as "attr", a method named "EOattr" is generated. The method returns an object of type EOObject.
(2) If the bound attribute is constructed through application operation in the source program, then the target code is placed into the method "EOattr" being generated. In this case the target code is generated as follows:
   (a) The method returns a new instance of the object being applied in the source program.
   (b) The instance is created through its constructor, where all its parameters are passed in the order of the order specified in the source program.
   (c) The evaluation strategy is down to the instance being returned (it decides how to evaluate itself on its own)
(3) If the bound attribute is constructed through abstraction operation in the source program, then a private inner class named "EOattr" is generated. The inner class is a subclass of the "EOObject" base class. The target code is generated as follows in this case:
   (a) Free attributes of the abstracted object are translated as described in subsection "EO Objects Free Attributes Translation".
   (b) Bound attributes of the abstracted object are translated as described in this section.
   (c) The method "EOattr" that wraps the "EOattr" private inner class returns a new instance of that class passing all arguments (free attributes) to it in the order specified according to the source program.
   (d) The evaluation strategy is implemented via overriding the "getDecoratedObject" standard method (as described in detail in the Dataization section)
   (e) To access the parent object, the "getParent" is used from EOObject (it is also overridden inside nested classes). It is done for simplicity and may change in the distant future when the compiler [3] becomes more feature rich.
   (f) When the source program explicitly accesses an attribute (e.g., EOattr) of the parent object (e.g., EOobj) of the attribute (e.g., EOboundAttr), it is translated to this.getParent().EOattr for simplicity as described in (c).

It is important to mention that memory leakage issues that inner classes are blamed for do not affect the proposed model performance compared to the original implementation of the compiler (current compiler) since the latter links all the nested objects (attributes) to their parents. As a further optimization, the transpiler may check if a bound attribute created through abstraction does not rely on the parent's attributes. In this case, the private nested class may be made static (i.e. not having a reference to its enclosing class). As it was mentioned in Naming & Scoping Rules, bound attributes (these are basically nested objects) are put to the scopes they originate from. So, bound attributes are translated into named private inner classes. This technique makes target code denser and also delegates the management of scopes and child-parent hierarchies to the Java Virtual Machine (JVM), not the Eolang runtime.

3.1.5 Anonymous Objects. The idea behind anonymous objects is simple. Every time an anonymous object is used in the source program (either in application to another object or in decoration), it is translated as an local class of the EOObject class right in the context it is originated from. All the principles of class construction take place in the local class just as in named classes (constructor generation, free and bound attributes translation, implementation of decoration & dataization mechanism, etc.).

3.1.6 Decoration and Dataization. Every user-defined object evaluation technique is implemented via an overridden "getDecoratedObject" standard method. The overridden method constructs the object's decoratee and delegates the object's own evaluation to the decoratee. The result of the evaluation of the decoratee is then returned as the result of the evaluation of the object itself. If the

decoratee is not present for the source program object, an exception is thrown inside the dataization getData method

3.1.7 Accessing Attributes in the Pseudo-Typed Environment. If the user code fragment utilizes an EOObject instance (and the concrete type is unknown), the Java Reflection API is used to access the attributes of the actual subtype. Since both free (inputs) and bound (outputs) attributes are wrapped as methods (that can have from zero to an arbitrary number of arguments) it is possible to handle access to all kinds of attributes through a uniform technique based on the dynamic method invocation Java Reflection API. The technique works as follows:

(1) Ask Reflection API to check if a method with the name EOattr and the necessary signature (with the correct number of arguments) exists. If it does not exist, throw an exception. Otherwise, proceed to step 2.
   (a) 1.1. If method not found, try to complete Point 1 for the getDecoratedObject().
(2) Invoke method passing all the arguments into it in the order specified.

3.1.8 Data Primitives and Runtime Objects. Data primitives and runtime objects also extend the EOObject base class. However, the main idea behind them is to make them as efficient as possible. To do that, data primitives objects are implemented in a simpler way compared to the guidelines of the proposed model. For example, bound attributes of the data primitive objects are implemented through methods only (no class nesting is used). Another important property of the runtime objects is that they are often atomic and, hence define their evaluation technique without a base on the decoratee as mentioned above. Instead, these dataize to the atomic data or perform a more efficient (semi-eager or eager) evaluation strategy.

3.2 Transformation from XML to Medium (intermediate) model

Some dependencies in the XML files of old version of compiler were noted:

(1) All abstractions, no matter their true location in the source "*.eo" file, are moved to the first nesting level. The translator makes the tree flat in terms of abstractions. In this way the translator assigns a name to all abstractions (including anonymous ones).
(2) All kinds of bound attributes, thus, abstraction-based & application-based bound attributes, within EO objects are essentially replaced by application in all cases. So, the application remains as application. And abstraction-based attributes turn into an application that refers to a previously nested object on the first nesting level.

Based on these two properties of the XML input document, it was decided to translate from XML to Medium as follows:

(1) Translate all abstractions. They are the first level of XML document nesting. Recursion is not applied at this stage, as abstractions are represented in a completely flat (linear) form.

Table 2: Runtime Model

Class	Description
EOObject	1.Instantiate an EO object (i.e. make a copy of it via a constructor). 2. Dataize an EO object 3. Get the parent object 4. Access attributes of an object 5. Get the decorated 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

(2) For each abstraction from step 1, all internal applications (i.e. bound attributes) are translated. This process is recursive because of the recursiveness of subtrees that belong to the appendices. After step 2, we have a list (array) of abstractions in the Medium model, each of which has a list of bound-attributes defined by applications, which are also translated into the Medium model

After that several optimizations are made by the resulting code model structure:

(1) Code Model Deflate Operation. This operation returns abstractions, which are "flat" (linear) in previous compiler model, to their places. Thus, the applications which defined abstraction-based attributes get a new field (wrappedAbstraction). Also the applications that referenced local abstraction-based objects get this field.
(2) Code Model Scope Correction. This operation corrects the Scope of all the Medium model entities. All applications have file, thus, the abstraction object in which they reside. All abstractions have Scope, thus, the parent abstraction object.The operations outlined above help the intermediate Medium code model become closer to the final model, and move away from the XML model. Both optimizations greatly simplify the final code translation from Medium model to Java.

The Eo-Java runtime model is summerized in Table 2 and the mapping of EO entities to Java code structures is displayed in Table 3.

4 RESULTS AND COMPARISON

4.1 Results

The main output of this work is the Eolang compiler, implemented based on the proposed model, and is now available at Github repository (OSS, MIT) [1]. The maven wrapper plugin is used to initiate compiling at the build phase to generate class files. A comparison of the new model with the old model is exhibited in Table 4. As shown in this table, the performance of the new model shows a big difference and major improvement in execution time. The target code

Table 3: EO Entities to Java

EO	Java
Abstraction	A plain old Java class extending EOObject. The plain old Java class may also be nested or local.
Application	1.Creating a new class instance 2. Direct method call 3.Calling the method via recursion
Package-scope application	Is wrapped by Abstraction because of the absence of package-levels in Java
A free attribute	A class field that should be set via the constructor
A bound attribute	A method that returns an object
Duck typing	Everything is EOObject. Class fields are accessed via Java Reflection
Object dataization	It is the default for all user-defined classes and relies on the getDecoratedObject()
getDecoratedObject()	To be overridden for evaluation strategy and used to access the attributes of the decoratee object
Anonymous EO objects	Implemented via Local Java Classes Nested class — attribute Local class — anonymous object

produced by the new runtime is more concise, less verbose, and shows potential for scalability. A sample of the generated source code is exhibited in the Appedix section.

Table 4: Comparison of execution time

Program	Current	Proposed
50th Fibonacci [16]	0m3,241s	0m0,191s
Pi Digits [17]	0m8,537s	0m4.023s
Factorial(20) [15]	0m4,628s	0m0.126s
Sum of array elements (50) [18]	0m5,111s	0m0.179s

5 DISCUSSION

5.1 Limitation

5.1.1 Anonymous objects must have no free attributes. In a situation where an object needs to be pass to, for instance, an array.map, it is expected to create an object that has an attribute with output parameters, as opposed to that of the previous runtime model.

This is due to the fact that an instance of the class cannot be created without passing the expected arguments. This happens where anonymous objects are used. In future, it may be possible to resolve this issue at the translator level so that the user code will not be affected by this limitation

5.1.2 Recursion now builds only in a lazy mode. What is impossible now: Use recursive definition inside a greedy construct. Greedy constructs include primitive methods like EOadd, EOmul, etc.

Example:

```
2.mul (recursive (n.sub 1))
```

It impossible now, because the recursion, due to greediness, will never be reduced to the result.

What is possible now: Use greedy approach inside recursive constructs

Example:

```
recursive (n.sub 1) (n.mul 2)
```

5.2 Possibilities for future development of the model

Now, with an intermediate code model in Java (i.e., an object tree built on the Medium model in Java), we can:

5.2.1 Add call optimization. There is a predictable and an unpredictable context.

(1) Predictable:

```
rectangle > rec
```

In this case, rec is a type-predictable variable. it is obvious that this is a type of some.package.name.here.EOrectangle. In a predictable context, we can opt out of recursion. Then

```
rec. attr 5
```

instead of rec.

```
_getAttribute ("attr", EOint(5L))
```

will be translated to

```
rec.EOattr(EOint(5L))
```

An increase in speed and even greater connectivity of objects is achieved. As a bonus: partial validation of the program by the Java compiler is in predictable contexts.

(2) Unpredictable context:

```
[a b] > rectangle
```

Here it is not obvious what [a] is. In unpredictable contexts, recursion is continually used. The unpredictable context is still one-access to the input attributes from inside the object. All other contexts are predictable

5.2.2 Add other possible optimization. If it's possible to simplify some object. Example:

```
[] > obj
    [] > @
        [] > @
            [] > @
                5.add 7 > @
```

The above can be simplify to:

```
5. add 7 > obj
```

5.2.3 Make the translator detect recursive call cases.

(1) It will try to replace normal recursion from object recursion (when objects are created recursively) to functional recursion (method chain). This will work faster and more efficiently but could also take a long time and resources.
(2) Tail recursion, the translator does not only covert into a functional chain, but also converts into a loop. Then the tail recursion in EO will be equivalent to a normal loop but that is even a more complex task than point 1.

6 CONCLUSION

We analyzed the current implementation of the Eolang compiler and pointed out the main issues. The issues were that the target code produced by the compiler was verbose and included many redundant lines, was less efficient compared to Java and showed limited potential for scalability. After taking all the above into consideration, a new model was proposed and implemented and the results were compared.

The new model (transpiler) is embedded into the pipeline of the Eo maven wrapper plugin. The maven wrapper plugin [4] is used to initiate compiling at the build phase to generate class files from Eolang source code based on the proposed runtime model, and then subsequently packaged.

Programs compiled with the new compiler have much better execution time than that of the current/previous compiler. The new compiler also has a great protential for scalability; the runtime is easier to maintain and it produces a more concise target code than the previous compiler. The new Eolang compiler is now ready and available at GitHub [1]. The proposed solution uses Java instead of XSLT[19] because XSLT is not scalable, too complex to comprehend, and can only transform one text to another or one xml fragment to another[9]. A scalable solution should, perhaps, have code that is clearly analyzable.

7 ACKNOWLEDGMENTS

This paper and the research behind it would not have been possible without the funding by Huawei Co. Ltd and the exceptional support of Eugene Popov, Joseph Afriyie Attakorah and Vitaliy Korzun.

8 APPENDICES

A FACTORIAL

```
package sandbox;

import org.eolang.*;
import org.eolang.core.*;
import java.util.function.Supplier;

/** Package-scope object 'appFactorial'. */
public class EOappFactorial extends EOObject {

  /** Field for storing the 'args' variable-length free
      attribute. */
  private final EOarray EOargs;

  /**
   * Constructs (via one-time-full application) the package-
       scope object 'appFactorial'.
   *
   * @param EOargs the object to bind to the 'args' variable-
       length free attribute.
   */
  public EOappFactorial(EOObject... EOargs) {
    this.EOargs = new EOarray(EOargs);
  }

  /** The decoratee of this object. */
  @Override
  public EOObject _getDecoratedObject() {
    return new org.eolang.txt.EOsprintf(
        new EOThunk(() -> (new EOstring("%d! = %d\n"))),
        new EOThunk(() -> (this.EOn())),
        new EOThunk(() -> (new EOfactorial(new EOThunk(() ->
            (this.EOn()))))));
  }

  /** Returns the object bound to the 'args' input attribute.
       */
  public EOarray EOargs() {
    return this.EOargs;
  }

  /** Application-based bound attribute object 'n' */
  public EOObject EOn() {
    return ((this.EOargs())._getAttribute("EOget", new
        EOThunk(() -> (new EOint(0L)))))
        ._getAttribute("EOtoInt");
  }
}
```

REFERENCES

[1] 2021. EO-lang compiler for simple cases. https://github.com/HSE-Eolang/eo
[2] Cardelli Luca Abadi, Martin. 1996. *A Theory of Objects.* Springer-Verlag New York.
[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.
[4] Raghuram Bharathan. 2015. *MApache Maven Cookbook.* Packt. https://www.packtpub.com/product/apache-maven-cookbook/9781785286124
[5] Yegor Bugayenko. 2017. *Elegant Objects.* https://www.elegantobjects.org/
[6] Yegor Bugayenko and Contributors. 2020. EO. https://github.com/cqfn/eo
[7] Amnon H. Eden and Yoram Hirshfeld. 2001. Principles in Formal Specification of Object Oriented Design and Architecture. In *Proceedings of the 2001 Conference of the Centre for Advanced Studies on Collaborative Research* (Toronto, Ontario, Canada) *(CASCON '01)*. IBM Press, 3.

[8] HSE-Team. 2021. *EO.* https://github.com/HSE-Eolang/eo/tree/4a4dc90eb7601ed24a044de4d5e8c79cc342812c
[9] Michael Kay. 2008. *XSLT 2.0 and XPath 2.0 Programmer's Reference, 4th Edition.* Michael Kay. https://www.wiley.com/en-us/XSLT+2+0+and+XPath+2+0+Programmer%27s+Reference%2C+4th+Edition-p-9780470192740
[10] Zeba Khanam. 2018. Barriers to Refactoring: Issues and Solutions.
[11] S. Nanz and C. A. Furia. 2015. A Comparative Study of Programming Languages in Rosetta Code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 778–788. https://doi.org/10.1109/ICSE.2015.90
[12] Oscar Nierstrasz. 1989. *A Survey of Object-Oriented Concepts.* Association for Computing Machinery, New York, NY, USA, 3–21. https://doi.org/10.1145/63320.66468
[13] Michael L. Scott. 2009. *Programming Language Pragmatics, Third Edition* (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[14] Michael L. Scott. 2015. Programming Language Pragmatics, Fourth Edition. https://www.elsevier.com/books/programming-language-pragmatics/scott/978-0-12-410409-9
[15] HSE Team. 2021. Factorial example. https://github.com/HSE-Eolang/eo/blob/master/sandbox/eo/app.eo
[16] HSE Team. 2021. Fibonacci example. https://github.com/HSE-Eolang/eo/blob/master/sandbox/eo/fibonacci.eo
[17] HSE Team. 2021. Pi digits example. https://github.com/HSE-Eolang/sandbox-examples/blob/main/eo/pi.eo
[18] HSE Team. 2021. Sum of array example. https://github.com/HSE-Eolang/sandbox-examples/blob/main/eo/sum.eo
[19] Doug Tidwel. 2008. *XSLT, 2nd Edition.* O'Reilly Media, Inc. https://www.oreilly.com/library/view/xslt-2nd-edition/9780596527211/