EO Programming Language Transcompilation Model for Java Source Code Generation

Hadi Saleh[*]
hsalekh@hse.ru
National Research University
Higher School of Economics
School of Software Engineering
Moscow, Russia

Evgeniy Popov
empopov@edu.hse.ru
National Research University
Higher School of Economics
School of Software Engineering
Moscow, Russia

Sergey Zykov[†]
szykov@hse.ru
National Research University
Higher School of Economics
School of Software Engineering
Moscow, Russia

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

ABSTRACT

The EO programming language is a novel initiative that aims to drive the course of elaboration of the proper practical application of the object-oriented programming paradigm. The EO language follows the eponymous paradigm advocating pure objects, free from incorrectly made design decisions common for mainstream technologies. EO is based on the formal model of object calculus—the ϕ-calculus. This work aims to enhance the implementation of the transpiler and the standard object library of the EO programming language. This study highlights essential features of the language and the underlying formal object calculus and proposes a translation scheme of EO programs to Java source code with a comparison of this scheme to an existing solution. The findings of this work are the proposed transcompilation model and a renewed transpiler implementing it. This work is valuable for the EO project as it facilitates the language to evolve and prepares it for further assessments on practical applicability, interoperability with Java, performance in enterprise applications, etc.

CCS CONCEPTS

• **Software and its engineering** → **Source code generation**; *Runtime environments*; *Object oriented languages*; • **General and reference** → *Design*; *Experimentation.*

[*]Also with Vladimir State University, Department of Information Systems and Software Engineering.
[†]Also with National Research Nuclear University Moscow Engineering Physics Institute, Institute of Cyber Intelligence Systems.

KEYWORDS

object-oriented programming, Java, the EO programming language, transcompilation, transpiler, the φ-calculus, elegant objects

ACM Reference Format:
Hadi Saleh, Evgeniy Popov, Sergey Zykov, and Alexander Legalov. 2021. EO Programming Language Transcompilation Model for Java Source Code Generation. In *ESEM '21: ACM Symposium on Empirical Software Engineering and Measurement, October 11–15, 2021, Bari, Italy.* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

Object-oriented programming (OOP) has been one of the dominant paradigms in the software development industry for the past two decades [7, 14]. Many technological enterprises and leading digital solution providers that utilize the mainstream OOP languages suffer from a lack of quality of their projects despite the tremendous effort and resources that have been invested in their development. Quality and maintainability issues might be explained by the essence of inherent flaws in the design of the programming languages and the OOP paradigm itself, as industry experts point out. Thus, drastically new languages and approaches have been developing to address the problem.

The EO programming language is one of the promising technologies that has arisen to drive the course of elaboration of the proper practical application of the OOP paradigm. The EO language emerged from the "Elegant Objects" paradigm that advocates the vision of pure OOP programming, free from the incorrectly made design decisions common for mainstream technologies [10, 11]. These are static code entities, inheritance, classes, mutable objects, null references, reflection, and global variables. EO is based on the formal model of object calculus—the φ-calculus [9]. The calculus defines four elemental operations as sufficient to describe object-oriented programming paradigm properties and semantics.

An existing implementation of the language is a transpiler that transforms EO programs to Java source code [4]. The transpiler has several problems, where the most noticeble one is the poor performance of the output Java code. This work aims to enhance the implementation of the transpiler and the standard object library of the existing implementation of the EO programming language through proposition and implementation of a new transcompilation model of EO programs to Java source code. Objectives of this paper are to analyze the language and the φ-calculus identifying essential parts of EO, to propose a transcompilation model of EO programs to Java source code, to compare the proposed model to an existing scheme, to probe the proposed translation model through implementation of a renewed EO transpiler and the standard object library of the language.

The findings of this work are the proposed transcompilation model and a renewed transpiler implementing it. The EO programming language is a research and development project that remains in an undeveloped state. Therefore, this work is considered valuable for the language to evolve and be prepared for further assessments on practical applicability, interoperability with Java, performance in enterprise applications, and other analyses and improvements.

This paper is structured as follows. Section 2 describes the EO language and its bases and summarizes the elemental parts of the language. Section 3 and 4 depict the existing and proposed transcompilation models of EO programs to Java source code qualitatively. Then, Section 5 compares performance of the new and the existing solutions quantitatively. Section 6 provides a discussion of the findings of this work with a summary of advantages and limitations of the proposed model. Finally, Section 7 concludes this paper and projects future work directions based on this research.

2 THE EO PROGRAMMING LANGUAGE

This section provides a concise description of the object of this study—the EO programming language. The description covers the "Elegant Objects" paradigm, the φ-calculus, the elements of the EO language, syntax and semantics of EO, and its relation to programming languages and paradigms.

2.1 The Elegant Objects paradigm

The EO language emerged from the "Elegant Objects" paradigm, which is a congregation of advised conventions to achieve proper object-oriented principles and designs in practice [10, 11]. The "Elegant Objects" paradigm advocates code free constructors, immutable objects, design by contract programming and does not admit null references, static code entities, reflective programming, type introspection at runtime, the pattern of getters and setters, and inheritance mechanism. The EO language—which is an attempt to form a small OOP language based on the paradigm—abides by most of the principles of the paradigm.

2.2 The φ-Calculus

The φ-calculus is a model of object calculus formulated by Y. Bugayenko as a proposition of the formal basis for the object-oriented programming paradigm and the EO language [9]. The φ-calculus vastly relies on the λ-calculus [13], applied in functional programming, as it defines objects—first-class entities of the φ-calculus—as sets of sub-objects (or, attribute objects), and the internals of atomic objects (meaning, non-reducible objects with the implementation details defined outside of the φ-calculus) as lambda terms. The φ-calculus defines abstraction, application, decoration, and dataization as the principal elements that comprise the sufficient set of operations that can describe object-oriented programming paradigm properties and semantics.

The φ- and λ-calculi are nearly cognate structurally and terminologically. The abstraction operation in either of the models is used to define new entities with input arguments and internal structures. In the λ-calculus, a function declared through abstraction can have only one argument and one internal lambda term determining the expression that the function reduces to [13]. In contrast, objects declared by means of the operation in the φ-calculus can have any number of free attributes (arguments or inputs) and bound attributes (outputs or objects structurally associated with their parent object).

Similarly, the operation of application in both calculi is used to substitute input arguments of entities. In the λ-calculus, only one term can be applied to another (although, currying technique enables functions to have multiple arguments [17, Section 2.2.1][13, Chapter 1, Paragraph 3]). In the φ-calculus, an arbitrary number of terms can be applied to an object being instantiated through application. Due to the hierarchical nature of objects, the φ-calculus additionally defines the dot-notation mechanism that allows application terms to access attributes of objects (including special cases, namely, parent objects, self-referencing, and decorated objects). The φ-calculus adapts the partial application technique used in functional programming. This technique enables terms of the operation of application to bind only a subset of arguments to free attributes of an object being instantiated, leaving some of attributes unbound.

The operation of object decoration defined in the φ-calculus has no direct counterpart in the λ-calculus. Nevertheless, it may be compared with the function composition mechanism widely employed in functional programming [18, Section 4.5.2]. As the function composition technique facilitates code factoring in the functional programming paradigm, so does the decoration operation in the φ-calculus. The operation allows an object to refer to another object—either through abstraction or application—as to the object it decorates. The instantiated link between the objects extends the set of attributes of the decorator object to the union of the initial set of its attributes and the set of attributes of the decoratee object. Thereby, the decoration operation performs the construction of the eponymous pattern utilized in object-oriented programming [15, Chapter 4]. Thus, decoration allows objects to be extended or, conversely, factored to maintainable parts.

The dataization operation performs the evaluation strategy over objects to extract data they represent. Dataization may be compared to the reduction strategy defined in the λ-calculus. While reduction operations accomplish a sequence of lambda terms substitutions, dataization performs recursive object tree evaluation due to the hierarchical structure of objects and, hence, applications of them. The dataization operation is declared as a call-by-need operation. Therefore, the base evaluation strategy in the φ-calculus is lazy in its nature.

Both φ- and classical λ-calculi do not define a type system as a part of them, leaving it to applications utilizing them—programming languages and compilers. So, a plain and simple implementation of the φ-calculus would be an untyped programming language. The absence of types would cause a need to determine the existence of referenced attributes of an object in runtime dynamically with no type checking mechanisms engaged. Such a language would comply with the tenets of the φ-calculus, however, it would have drawbacks—produced code would be unsafe and operating slower comparing to compiled programming languages. A typed φ-calculus might be introduced through research in the future, similar to several typed λ-calculi developed [8, 12, 16]. However, this problem is out of the scope of the research questions of this paper, so it will not be considered hereafter.

2.3 Fundamental Elements, Syntax, and Semantics of EO

This section covers the elemental parts, syntactical and semantical properties of the EO programming language [9]. These include objects, attributes, the four operations defined in the φ-calculus, and the type system.

Listing 1 demonstrates an example program in EO. The program comprises two package-level objects: `pi` and `circle`. This section shows the elements of the language relying on this example.

Listing 1: An example code written in EO

```
1  # The Pi number (approx.)
2  3.1415926 > pi
3
4  # Represents a circle
5  # with a center point (x,y)
6  # and a radius
7  [radius x y] > circle
8    [] > @
9      ^.radius > @
10
11   # The circumference of this circle
12   mul. > circumference
13     2.0
14     mul.
15       pi
16       radius
17
18   # The area of this circle
19   pi.mul (radius.mul radius) > area
20
21   # Determines whether the point
22   # (x,y) is inside this circle
23   [x y] > isInside
24     leq. > @
25       add.
26         (x.sub (^.x)).pow 2.0
27         (y.sub (^.y)).pow 2.0
28       (^.radius).mul (^.radius)
```

2.3.1 Objects and Attributes. An object—the centric notion of EO—is a set of attributes. Every entity in the EO language is an object. An object can have an arbitrary number of free and bound attributes. Attributes of objects—and, hereby, objects as well—are immutable, meaning attributes can be associated with corresponding objects only once, and no modifications are allowed. Objects with at least one free attribute are abstract, and those with no free attributes are closed.

Every object has a scope it belongs to. An object may be scoped to a package (package-level scope, for instance, objects `pi` and `circle` at lines 2 and 7 of Listing 1), to another object declared through abstraction (attribute-level scope, for instance, objects `area` and `isInside` at lines 19 and 23 of Listing 1 are attributes of the `circle` object), or to an application term (application-level or anonymous scope. for example, an anonymous object at line 8 of Listing 1 is bound to the `@` attribute of the `circle` object). Scope is declared structurally (i.e., by the context an object is declared in) and cannot be changed dynamically. An object accesses other objects through its scope. There are four types of object access in EO:

- Accessing the parent object. This type of access is always explicit only, so to reference the parent object or an attribute of it a programmer must use the special `^` identifier. Line 26 of Listing 1 shows an example of accessing the parent object. Here, `x` refers to the `x` free attribute of the `isInside` object, and `^.x` refers to the attribute with the same name of the parent `circle` object.
- Accessing objects in the decoration hierarchy. This type of access is implicit by default and may be denoted unambiguously through the special `@` identifier. Due to semantics of decoration (see Decoration section below), an implicitly referenced attribute in the decoration hierarchy may be shadowed by an attribute with a similar name defined in the object itself.
- Accessing the object itself or one of its attributes. This type of access is implicit by default and may be denoted unambiguously through the special `$` identifier.
- Accessing an object from an outer scope (package level objects referencing). If an accessed object is declared outside of the file where the reference appears, it is accessed through the alias name given on the file level.

Attribute access in any scope is performed through the dot notation.

2.3.2 Abstraction and Application. Abstraction and application allow a programmer to create objects. However, they operate differently. Structurally new objects can be declared through abstraction. In other words, declaration of free attributes as well as the internal structure of an object is possible by means of abstraction only. Application is used to instantiate an object with binding arguments to its free attributes.

Both operations are hierarchical and recursive. Abstraction may define an arbitrary number of inner attribute objects based on abstraction and application, and one application term may have other applications (and anonymous abstractions) inside it. This property, thus, makes EO programs structurally (or declaratively) hierarchical.

2.3.3 Decoration. The decoration operation defined in the φ-calculus can be performed over an object in a declarative manner through binding its special `@` attribute to a language expression (either abstraction or application) denoting a decorated object as demonstrated at lines 8, 9, and 24 of Listing 1. A decorator object inherits all the attributes of its decoratee and may define its own attributes including those that shadow some of the attributes of the decorated object. Shadowing is a semantical property of the language that defines rules of resolving names when similar identifiers are declared in different scopes. Attributes of a decorator object shadow those of a decorated object when their names and argument signatures are identical.

2.3.4 *Dataization.*

As described in section 2.2, dataization defines the evaluation scheme of EO programs. For all programmer defined objects, dataization relies on corresponding decorated objects entirely. To put it more simply, objects defined in EO programs delegate their evaluation scheme to their decoratees. In contrast, atomic objects (i.e., defined and implemented outside of the EO environments, for instance, objects of the standard library) may declare their own evaluation strategies.

This property of dataization implies several notable characteristics of the language. First, due to the lazy nature of the decoration operation defined in the φ-calculus, all programmer-defined objects have a lazy evaluation strategy as well. However, atomic objects may control their evaluation strategy freely making it eager or lazy in different contexts. Second, since dataization is the evaluation mechanism of EO and it relies on decoration completely, a program written in EO is decomposed to a set of objects, and an entry point object of the program defines its evaluation path through decoration by binding other objects to the `@` attribute (while these denote their evaluation schemes in the same manner). As a result, EO programs are evaluated hierarchically, too. In fact, an overall dataization strategy of any object or a set of objects may be denoted structurally as a general tree (meaning, each node may have an arbitrary number of child nodes). Moreover, some of objects in the evaluation tree must delegate their dataization scheme to atomic objects to make the tree reducible. Finally, programmer-defined objects with no decoratees may not be evaluated.

2.3.5 *Type System.*

The EO programming language is an untyped language. EO has neither explicit type declaration syntax, nor type inference mechanisms. The hierarchical multi-faceted nature of objects causes a need to determine existence of referenced attributes of an object in runtime dynamically. However, types of objects may be inferred by implementations of the language in some cases—when a type of an object may be determined directly from the source code—and, thus, primitive type checking may take place.

2.3.6 *Relation of EO to Programming Languages and Paradigms.*

Since the EO programming language is based on the novel formal model of the φ-calculus showing resemblance with its functional programming equivalent—the λ-calculus—and follows the principals of the "Elegant Objects" paradigm, which renounces traditional OOP techniques, the language might be categorized to either of functional or object-oriented paradigms. However, EO is not a functional language since it has objects, not functions, as its first-class citizens. The φ-calculus rather supersets and redefines the λ-calculus in terms of objects, which is why the language may not be classified as functional. On the other hand, "Elegant Objects" principles backing the language shrinks commonly used OOP features to a restricted subset of them. This obstructs classifying the EO language from the object-oriented programming paradigm in its commonly appreciated vision directly as well. However, EO may be categorized as a declarative object-oriented programming as it operates over objects—its primary and only entities—declaratively.

3 THE EXISTING IMPLEMENTATION OF THE LANGUAGE

This section describes the transcompilation model of EO programs to Java source code (as well as its implementation in a form of a transpiler) proposed by Yegor Bugayenko, the main contributor of the EO programming language project [4]. Further in this section, the model is referred to as existing or current.

3.1 The Transcompilation Model

The existing model defines a transcompilation scheme of EO programs to Java source code. This subsection summarizes the mapping rules for each elemental part of the language defined within the scheme.

3.1.1 *Objects Definition Through Abstraction.*

Objects declared in EO programs by means of the abstraction operation are mapped to Java classes extending the `PhDefault` base type that defines the standard internal structure of all objects in the EO runtime environment, specifically the mechanism of objects cloning and the apparatuses of storing, retrieving, and mutating attributes of objects.

Attributes of an object are stored within an associative array inside the object. Keys of elements of the array are names of the attributes of the object, while values are instances of classes of the EO standard library denoting the contents of the attributes—corresponding EO objects. Target Java classes resulted from translation of EO objects declared through abstraction contain the constructor that assembles the associative array defining free and bound attributes of corresponding EO objects. A free attribute of an object is stored as an instance of the `AtFree` EO runtime standard class (meaning, the attribute may be substituted with a concrete value) and a bound attribute is stored as an instance of the `AtBound` class containing a lambda expression that defines the contents of the bound attribute. Listing 2 shows a simplified example of this structuring principle.

Listing 2: Internal structure of a target class produced by the existing model

```java
public EOcircle(final Phi parent) {
  super(parent);
  this.add("radius", new AtFree(/* default */));
  this.add("x", new AtFree(/* default */));
  this.add("y", new AtFree(/* default */));
  this.add("isInside", new AtBound(new AtOnce(
  new AtLambda(this, self -> {
    Phi ret = new EOcircle$EOisInside(self);
    ret = new PhCopy(ret);
    return ret;
11   }))));
12   /* other attributes declarations */
13 }
```

The associative array containing the attributes of the object may be mutated through the operation of addition of new attributes. Due to the mutable nature of the array, the EO runtime includes the `PhCopy` object that is a cloning utility class instantiating an exact copy of the object being cloned. This class is used to ensure the characteristic of immutability of objects when they are copied or passed as arguments through the operation of application. An illustration of usage of the cloning class is shown at lines 4 and 7 of Listing 4.

As described in Section 2.3.1, every object is declared in one of the scopes:

- Package-level scope.
- Attribute-level scope.
- Application-scope (or anonymous scope).

In any case, objects declared through abstraction are translated to public package-level Java classes, and each Java class is stored in a separate file. Every separated file is kept within a Java package with a name identical to the one declared through the `package` meta directive at the top of the EO source file the abstracted object is stored in. Nested and anonymous EO objects are flattened out and stored outside the original scopes they belong to. To mitigate naming conflicts that may be potentially caused due to the flattened nature of the target source code, the existing model encodes the original scopes of objects in names of their Java files and classes delimiting parts of the names with a dollar sign symbol. Listing 3 demonstrates how the anonymous object bound to the `@` attribute of the `circle` object (see line 8 of Listing 1) is transpiled to a separate Java class with the naming technique applied to differentiate its original scope.

Listing 3: Translation of anonymous abstraction in the existing model

```
1  public final class EOcircle$EOφ extends PhDefault {
2    public EOcircle$EOφ(final Phi parent) {
3      super(parent);
4      this.add("φ", new AtBound(new AtOnce(
5      new AtLambda(this, self -> {
6        Phi ret_base = new PhMethod(self, "ρ");
7        Phi ret = new PhMethod(ret_base, "radius");
8        return ret;
9      }))));
10   }
11 }
```

3.1.2 *Objects Instantiation Through Application.*

An application term—that instantiates an object providing its free attributes with concrete values—associated with a bound attribute of some object is translated to a Java lambda expression contained in an instance of the `AtBound` class stored in the attributes associative array of the object. This lambda expression contains statements denoting the actual contents of the application term. The application term, as stated in Section 2.3.2 may hierarchically include other application terms. These are translated as Java statements placed within the same lambda expression.

The Java statements that the application term is translated to may consist of the following parts:

- Class instantiation. This part is used when an object defined through abstraction is applied. The instantiated object is then copied and (optionally) its free attributes are provided with concrete values (see lines 6-9 of Listing 2).
- Attribute access through creation of the `PhMethod` class instance. This part is used when an attribute of an object is applied (including special attributes `@`, `^`, `$`, as described in Section 2.3.1). Lines 3, 5, 6, and 8 of Listing 4 illustrate this.
- Free attribute binding through creation of an instance of the `PhWith` class. This part is used when free attributes of an object are provided with concrete values through the application term (see lines 9 and 10 of Listing 4).
- Object cloning through creation of an instance of the `PhCopy` class. This part is used to ensure the immutability characteristic of EO objects when mutating attributes. An example of the part is shown at lines 4 and 7 of Listing 4.

Listing 4: Translation of application in the existing model

```
1  this.add("area", new AtBound(new AtOnce(
2  new AtLambda(this, self -> {
3    Phi ret_base = new EOpi(self);
4    Phi ret = new PhMethod(ret_base, "mul");
5    ret = new PhCopy(ret);
6      Phi ret_1_base = new PhMethod(self, "radius");
7      Phi ret_1 = new PhMethod(ret_1_base, "mul");
8      ret_1 = new PhCopy(ret_1);
9        Phi ret_1_1 = new PhMethod(self, "radius");
10       ret_1 = new PhWith(ret_1, 0, ret_1_1);
11     ret = new PhWith(ret, 0, ret_1);
12   return ret;
13 }))));
```

3.1.3 *Objects Decoration.*

As described in Section 2.3.3, decoration in EO is performed in a declarative manner through binding the special `@` attribute of a decorator object to a language expression (either abstraction or application) denoting a decorated object. So, internally, the constructor of the target Java class of the decorator object appends a new element to the attributes associative array of the class. The appended element has the key `φ` and a value that corresponds to the decorated object. This is shown at lines 4-8 of Listing 3.

3.1.4 *Dataization Strategy of Objects.*

Dataization of EO objects defined by programmer relies on the decoration operation. Transpiled Java source code does not perform any actions unless it is demanded since the actual code is placed within lambda statements inside instances of the `AtLambda` class, as shown at line 1 of Listing 4. Once dataization of an object is started, the evaluation tree is built and traversed down to atomic objects defined in the standard library. The standard EO objects are lazy. Thereby, dataization strategy of all objects is lazy.

3.1.5 Typing. Attribute access of objects is done through the `PhMethod` class instantiation (see line 3 of Listing 4). This class performs attribute access through a lookup in the attribute associative array of the object. If the object does not have the referenced attribute, the evaluation of the program fails. Otherwise, the referenced attribute is returned.

Free attribute binding is done through the `PhWith` instance creation (see line 9 of Listing 4). This class performs a lookup of the referenced free attribute in the attribute associative array of the object. If the object does not have the referenced attribute, the program fails. Otherwise, the referenced attribute is bound.

All objects in the EO runtime environments have the same Java type. From the Java Virtual Machine perspective, all EO objects have the same type with an identical set of fields, methods and constructors. Hence, no compile-time type checks are performed. As showed above, all verifications are done at runtime through lookups of the dynamically formed attributes associative array. Thus, the existing model has no type system.

3.2 Problems of the Existing Model

The existing transcompilation model has several problems:

(1) High machine resource consumption. The model produces a lot of instances at runtime, since it clones objects to ensure their immutability and simulates the semantics of EO through standard classes instantiation (e.g., `AtFree`, `PhWith`, `PhMethod`—each of these objects simulates a sole property of EO semantics). This results in low performance of the target code that cannot perform even the easiest tasks, for instance, array sort or calculation of Fibonacci sequence numbers, as shown in Section 5.
(2) Low readability of the target code. Although the target code may be composed and structured in any manner, readability is important since one of the reasons to make the existing implementation of the EO language in a form of an EO-to-Java transpiler was an ability to examine and debug the target code. The existing model produces verbose code scattered to separate files, so a programmer that reads the target code needs to assemble disjointed parts of EO objects together to comprehend the entire picture the code depicts. Moreover, the target code consists of codified local variables and obfuscated operations over these variables and over instances of the EO runtime, so a programmer needs to match pieces of the target code with source EO objects and decipher the meaning of the pieces to locate the piece that is of interest.
(3) Code redundancy. The existing transpiler produces redundant code by cloning objects to ensure their immutability in cases when copying is not necessary (i.e., when the object is closed or when the object has no free attributes). In addition, the target code for the application operation declares a lot of local variables, although it is not necessary for general.

3.3 Implementation of the Existing Model

The software implementing the existing model is available at [4]. The project consists of the following modules:

- `eo-parser`: performs the initial parsing of EO code and produces XML documents that describe parsed programs.
- `eo-runtime`: collects objects of the standard object library and classes of the runtime environment (e.g., `PhDefault`, `AtFree`).
- `eo-maven-plugin`: supplies a plugin for Maven that translates EO programs to Java source code.

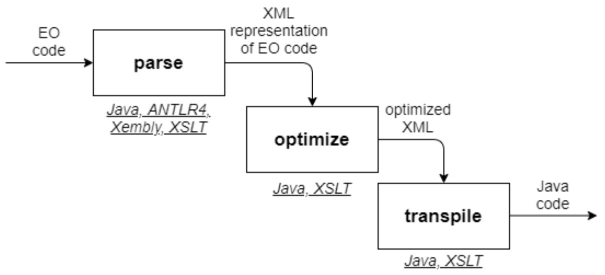

Figure 1: The pipeline of the existing transpiler

Figure 1 demonstrates the processing scheme of the existing transpiler, with inputs (on the left side of each rectangle), outputs (on the right side), and used technologies for each of the stages (at the bottom of rectangles). The transpiler relies on Java, ANTLR4 parsers generator toolkit, Xembly XML processing library, and XSL transformations of XML documents representing EO programs. Virtually, the existing transpiler performs transcompilation through transformations of XML documents. The stages of the transcompilation process are as follows:

(1) Parsing of EO code with ANTLR4 and construction of XML files that describe the code utilizing Xembly.
(2) Optimization of constructed XML documents. Optimizations are XSL transformation that assess programs against the rules of the language and prepares XML documents for the following stage.
(3) Producing Java code through XSL transformations.

4 THE PROPOSED TRANSCOMPILATION MODEL

This section describes the transcompilation model of EO programs to Java source code (as well as its implementation in a form of a transpiler) proposed in this paper. In this section, the model is referred to as the proposed model.

4.1 The Transcompilation Model

Objects declared in EO programs by means of the abstraction operation are mapped to Java classes extending the `EOObject` base type that defines the standard internal structure of all objects in the EO runtime environment, specifically the mechanism of retrieving data and accessing attributes of objects.

4.1.1 Objects Definition Through Abstraction. An EO object of the package-level scope is translated into a Java class stored in a separate file. EO objects of attribute level and anonymous scopes are not present as separate Java files. Instead, these are put to the scopes they are originated from. So, abstraction-based attributes of an object are translated to private inner classes [2], and anonymous EO objects are translated as in-place local Java classes placed directly within methods they are used in [1]. This technique preserves original structure of objects and delegates management of scoping and parent hierarchies to the Java Virtual Machine. Listing 5 demonstrates a simplified example of the proposed structure, where the `EOcircle` public package-level class defines the internal structure of the original `circle` object, and the `EOisInside` private nested class represents the source `isInside` attribute object. An example of a local class representing a source anonymous EO object is demonstrated in Listing 6.

Listing 5: Internal structure of a target class produced by the proposed model

```
1  public class EOcircle extends EOObject {
2    private final EOObject EOradius;
3    private final EOObject EOx;
4    private final EOObject EOy;
5
6    public EOcircle(EOObject EOradius, EOObject EOx,
     ↪  EOObject EOy) {
7      this.EOradius = EOradius;
8      this.EOx = EOx;
9      this.EOy = EOy;
10   }
11   public EOObject EOradius() {
12     return this.EOradius;
13   }
14   public EOObject EOx() {
15     return this.EOx;
16   }
17   public EOObject EOy() {
18     return this.EOy;
19   }
20   public EOObject EOisInside(EOObject EOx, EOObject
     ↪  EOy) {
21     return new EOisInside(EOx, EOy);
22   }
23   private class EOisInside extends EOObject {
24     /* here is a regular class structure */
25   }
26   /* other attributes declarations */
27 }
```

The target Java class (of any scope) has only one constructor. There are two possible cases:

(1) If the source EO object has no free attributes, the resulting constructor has no arguments as shown at line 4 of Listing 6.
(2) If the source EO object has free attributes, the resulting constructor has all the arguments in the order of their appearance in the source program. All arguments of the constructor are of type `EOObject`. An illustration of this is depicted at lines 6-10 of Listing 5. If free attributes are present, the default constructor is disabled (which is the default Java semantical behavior).

For each free attribute of the source EO object, the following target Java entities are generated:

(1) A private final class field of type `EOObject` (see lines 2-4 of Listing 5).
(2) A public wrapper method of return type `EOObject`. The method returns the corresponding field. Lines 11-19 of Listing 5 show this.
(3) An argument in the sole constructor (in the order of appearance of the free attribute in the source EO object). The body of the constructor sets the corresponding field to the value of the argument.

Bound attributes of the source EO object are translated as follows:

(1) For every bound attribute, a wrapper method of the `EOObject` return type is constructed.
(2) If the bound attribute is constructed through the application operation in the source EO program, then the target Java code denoting the application term is placed into the wrapper method as shown in Listing 7.
(3) If the bound attribute is constructed through the abstraction operation in the source EO program, then a private inner class is generated as shown at lines . The translation scheme for attribute objects is the same, except an overridden version of the `_getParentObject` method is generated (see lines 5-8 of Listing 6). This method returns a reference to the parent object of the attribute object. The wrapper method returns a new instance of the generated inner class passing arguments (free attributes) to it in the order of their appearance in the source EO program. Lines 20-22 of Listing 5 demonstrate a wrapper method for a bound attribute constructed through abstraction, and lines 23-25 show a simplified version of a private nested class (its internal structure nearly replicates the structuring principles of package-level abstractions).

Listing 6: Translation of anonymous abstraction in the proposed model

```
1    @Override
2    public EOObject _getDecoratedObject() {
3      class anonymous$1 extends EOObject {
4        public anonymous$1() {}
5        @Override
6        public EOObject _getParentObject() {
7          return EOcircle.this;
8        }
9        @Override
10       public EOObject _getDecoratedObject() {
11         return _getParentObject()
12               ._getAttribute("EOradius");
13       }
14     }
15     return new anonymous$1();
16   }
```

4.1.2 *Objects Instantiation Through Application.*

An application term—that instantiates an object providing its free attributes with concrete values—associated with a bound attribute of some object is translated to a Java expression placed in the corresponding wrapper method. The application term, as described in Section 2.3.2 may hierarchically include other application terms. These are translated to subexpressions contained within the parent expression.

The Java expression that the application term is translated to may consist of the following parts:

- Constructor call. This part is used when a package-level object is applied (see line 2 of Listing 7). Substitutes of free attributes of the applied object are passed to the constructor.
- Attribute access through the `_getAttribute` method call (see line 3 of Listing 7). Alternatively, the access may be performed through a plain Java method call when referencing is done in the self-scope of the target class, as shown at line 6 of Listing 7. This part is used when an attribute of an object is applied. Objects to bind to free attributes of the applied attribute object are passed to the method.

All arguments to constructor and method calls are wrapped with an instance of the special class `EOThunk` as illustrated at line 4 of Listing 7. The actual argument is stored in a lambda expression inside the thunk object. The thunk object unwraps its contents once any message is sent to it (in other words, it implements the call-by-need evaluation scheme). This technique is used to avoid the eager evaluation scheme of arguments—that contradicts the semantics of EO—embedded into Java.

Listing 7: Translation of application in the proposed model

```
1 public EOObject EOarea() {
2   return (new EOpi())
3       ._getAttribute("EOmul",
4           new EOThunk(
5           () ->
6           ((this.EOradius())
7               ._getAttribute("EOmul", new EOThunk(()
                ↪  -> (this.EOradius())))))));
8 }
```

4.1.3 Objects Decoration. As described in Section 2.3.3, decoration in EO is performed in a declarative manner through binding the special `@` attribute of a decorator object to a language expression (either abstraction or application) denoting a decorated object. Hence, the decoration operation is translated in the same way as all bound attributes, except an overridden version of the standard `_getDecoratedObject` method is generated as the wrapper as shown in Listing 6.

4.1.4 Dataization Strategy of Objects. Dataization of EO objects defined by programmer relies on the decoration operation. Transpiled Java source code does not perform any actions unless it is demanded since the actual code is placed within lambda statements. Once dataization of an object is started, the evaluation tree is built and traversed down to atomic objects defined in the standard library. Some of the standard EO objects are eager (specifically, ones that perform arithmetical computations). So, dataization strategy of the proposed model depends on the objects used. It may be either eager or lazy.

4.1.5 Typing. Attribute access of objects is done through the standard `_getAttribute` method that utilizes the Java Reflection API. The API dynamically lookups the list of methods of the object. If the object does not have the referenced method, the evaluation of the program fails. Otherwise, the referenced object is returned.

Free attribute binding is done through passing arguments to constructors or methods of classes. If the call signature does not correspond to the one declared in the callee, the program fails. However, this verification is rather synthetical since it does not check whether the passed objects would match the callee internal structure semantically. In the case when the object being instantiated is an attribute object of an object of type `EOObject` (meaning, the actual type cannot be inferred), the Java Reflection API is used to instantiate it dynamically.

All objects in the EO runtime environments have the same Java type `EOObject`. From the Java Virtual Machine perspective, all of the objects have the same type with an identical set of fields, methods and constructors. Thus, no compile-time type checks are performed. As showed above, most verifications are done at runtime through Java Reflection API dynamically. Thus, the proposed model has no type system.

4.2 Problems of the Proposed Model

As described in Sections 4.1.1 and 4.1.2 constructors and methods used to instantiate objects through the application operation have necessary parameters of the same `EOObject` type. Practically, this means that the proposed model does not support the partial application mechanism of EO mentioned in Section 2.3.2. Moreover, the proposed model relies on the order of appearance of arguments in the source EO program. However, the language has a special syntax to bind free attributes by name, not by their order. So, the model does not support this mechanism as well.

4.3 Implementation of the Proposed Model

The software implementing the proposed model is available in [5]. The source project of the model implementation replicates the original module structure, transcompilation pipeline, and Maven plugin deliverable for convenient distribution of the transpiler. However, the proposed solution redefines several modules of the project and stages of the pipeline. So, the renewed transpiler uses output XML documents of the optimization stage of the original solution to reconstruct the tree of the source program in the form of Java (and Kotlin) runtime objects. After it, the object tree is transpiled to Java target platform hierarchically. We decided to reject XSLT and perform transcompilation using Java and Kotlin in order to perform more complex optimizations, checks, and analyses of EO programs in future work.

5 APPROBATION OF THE PROPOSED MODEL

This section provides the report on the conducted approbation experiments aiming to measure and compare the performance of the existing and proposed models.

5.1 Experiment Design

The approbation experiment was designed as follows. The output target Java sources produced by the implementations of the existing and the proposed transcompilation models for several algorithms implemented in EO (specifically, recursive factorial, array merge sorting, tail-recursive Fibonacci) were benchmarked by means of the "Java Microbenchmark Harness" (JMH) micro benchmarking utility provided by Oracle [3]. The benchmarked quantity evaluated in the experiment was time of execution of one algorithm run for each model. All the algorithms used in the experiment relied on recursion, so both models used stack hugely. For this reason, JMH was allocated with 64 megabytes of stack for each tested model. Such an amount of memory was used to benchmark as complex tasks as possible to retrieve representable data.

5.2 Performance Comparison of the Models

Table 1 shows the results of the experiment. As the measurements show, the code produced by the proposed transcompilation model implementation is more performant than the target code generated through the existing transpiler. This advantage in the performance of the proposed model is observed in all considered cases and all the tested algorithms. It may be explained by the following differences between the models:

- The existing transpiler produces chains of nested instances of runtime classes (e.g., `AtFree`, `AtBound`, `PhMethod`, amongst others mentioned in Section 3.1) to construct an EO-compliant object in the target Java platform. In contrast, the proposed model generates plain Java classes designed to abide by EO semantics, and no chains of instances are produced. This clearly improves the performance of the proposed model as heap allocations and runs of the garbage collector decrease.
- As described in Section 3.1.2, the existing model ensures the immutability characteristic of EO objects through copying them in any case when their attributes may be mutated. The proposed model refuses the partial application mechanism of EO, and conforms the immutability principle by making all arguments necessary in application of objects. This rids the runtime environments of a lot of copies of objects and, hence, makes programs more performant.
- While the evaluation strategy of the existing runtime is completely lazy, the proposed model makes—when it is possible—some objects of the standard library more eager. This augments the total performance of target Java code, too.

6 DISCUSSION

The proposed model abides by the principal properties of the semantics of EO. The model is fully immutable since a code fragment that applies (or copies) a class can access one and the only constructor and all arguments of the constructor are required. Fields that store free attributes of the object are final. The default constructor is disabled. Once sat, an attribute cannot be changed. Bound attributes are translated to methods (and, in some cases, inner classes). These Java source code entities cannot be changed. Because of that, objects are fully immutable without proposing cloning techniques utilized in the existing model. The model is dynamically-typed as it accesses attributes of objects through the Java Reflection API at runtime. These properties of the model were achieved by utilizing plain Java code structures and semantics in a concise manner.

Table 1: Execution Times of Code Produced by Models

Algorithm	n	Existing, *ms*	Proposed, *ms*
Recursive factorial of ***n***	1	1.41	0.09
	10	14.71	0.14
	100	1244.75	0.54
	1000	124462.43	3.62
	10000	—	36.56
	100000	—	442.22
Merge sort of an array of length ***n***	1	0.92	0.24
	2	21.48	7.32
	3	1257.56	91.69
	4	1857.67	262.47
	5	432938.08	3386.93
	6	—	8764.95
	7	—	15167.25
Tail-recursive Fibonacci ***n***th term	1	2.2	0.12
	10	11.69	0.14
	100	—	0.53
	1000	—	3.64
	10000	—	37.18
	100000	—	356.94

The proposed model addresses the problems of the existing solution listed in Section 3.2. The target code is structured and scoped exactly as the source EO program. The target code is commented by the renewed transpiler. The code neither declares local variables, nor does it simulate the semantics of EO through instances of the runtime classes, and the runtime consists of only three classes—`EOObject`, `EOData`, and `EOThunk`. Moreover, the transpiler formats the code so that it is more perceivable for the reader. Therefore, the proposed model produces more readable code than the existing solution. The code is more performant as well because of the model design decision taken.

The model has two limitations regarding the semantics of the application operations, as described above in Section 4.2. These problems will be addressed in future work. A possible solution for them is as follows. Java does not support named and default parameter techniques. Constructor and method overloading relies on signatures of overloaded procedures, and not on names of their parameters. Since all parameters have the same `EOObject` type, overloading cannot be used to solve the limitations of the model. The proposed model may utilize the Builder pattern to address the problems with partial application and named binding. This pattern allows one to instantiate objects more flexibly avoiding some arguments or changing their order.

In addition, in future work, a typed implementation of EO will be considered. The pseudo-type system to be proposed would infer types of objects in the target Java platform in determined contexts only. Determined contexts are code scopes where the type of an object is known and cannot be changed. Indeterminate contexts are those where the type of an object is dynamic and cannot be inferred. An example of indeterminate typing context (where a type may not be inferred) is referencing attributes of free attributes of an object. Semantically, free attributes of objects must be typed dynamically to keep the flexible nature of the application operation. Type inference on the target Java platform would increase the performance of the output code and enable basic type verification at compile time.

7 CONCLUSION

To conclude, this work proposes a transcompilation model of EO programs to Java source code, offers a renewed EO transpiler and the standard object library implementing the proposed model, compares the proposed and the existing models, and benchmarks their performance on several algorithms to show the achieved enhancement. The findings of this work facilitate the EO language to evolve from an undeveloped state and to be prepared for further assessments on practical applicability, interoperability with Java, performance in enterprise applications, and other analyses and improvements.

In future work, the limitations of the proposed model will be addressed. In addition, a typed implementation of EO will be considered. Moreover, a practical approbation of the language—through implementation and integration of an EO module in the jPeek Java code metrics measurement toolkit [6]—will be performed.

ACKNOWLEDGMENTS

This paper and the research behind it were funded by Huawei Technologies Co., Ltd. (TC202012080007) and supported by Yegor Bugayenko, Ivan Spirin, Maksim Shipitsin, Joseph Afriyie Attakorah, and Vitaliy Korzun.

REFERENCES

[1] Oracle Corporation 1995, 2021. *Local Classes.* Oracle Corporation. Retrieved May 10, 2021 from https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html
[2] Oracle Corporation 1995, 2021. *Nested Classes.* Oracle Corporation. Retrieved May 10, 2021 from https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
[3] The OpenJDK Community 2013–2021. *Java Microbenchmark Harness (JMH).* The OpenJDK Community. Retrieved May 10, 2021 from https://github.com/openjdk/jmh
[4] The EO Community 2016–2021. *EOLANG.* The EO Community. Retrieved May 10, 2021 from https://github.com/cqfn/eo
[5] The EO Community and the Team 2016–2021. *EOLANG.* The EO Community and the Team. Retrieved May 10, 2021 from https://github.com/hse-Eolang/eo
[6] The jPeek Community 2017–2021. *jpeek.* The jPeek Community. Retrieved May 10, 2021 from https://github.com/cqfn/jpeek
[7] TIOBE 2021. *TIOBE Index: Very Long Term History.* TIOBE. Retrieved May 10, 2021 from https://www.tiobe.com/tiobe-index/
[8] Henk Barendregt. 1993. Lambda Calculi With Types. In *Handbook of Logic in Computer Science*, D.M. Gabbay S. Abramsky and T.S.E. Maibaum (Eds.). Vol. 2. Oxford University Press, New York, NY, 117–309. https://doi.org/10.5555/162552.162561
[9] Yegor Bugayenko. [n.d.]. EOLANG and ϕ-calculus. ([n. d.]). Retrieved May 10, 2021 from https://github.com/cqfn/eo/tree/master/paper work in progress.
[10] Yegor Bugayenko. 2017–2021. *Elegant Objects.* Retrieved May 10, 2021 from https://www.elegantobjects.org/
[11] Yegor Bugayenko. 2020. *Elegant Objects* (1.7 ed.). Vol. 1. CreateSpace, Palo Alto, CA, United States.
[12] Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic* 5, 2 (June 1940), 56–68. https://doi.org/10.2307/2266170
[13] Alonzo Church. 1985. *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies).* Princeton University Press, Princeton, NJ, United States.
[14] KellyAnn Fitzpatrick. 2021. *RedMonk Top 20 Languages Over Time: January 2021.* RedMonk. Retrieved May 10, 2021 from https://redmonk.com/kfitzpatrick/2021/03/02/redmonk-top-20-languages-over-time-january-2021
[15] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* (1st ed.). Addison-Wesley, Boston, MA, United States.
[16] Masahito Hasegawa. 1995. Decomposing Typed Lambda Calculus into a Couple of Categorical Programming Languages. In *Proceedings of the 6th International Conference on Category Theory and Computer Science (CTCS '95).* Springer-Verlag, Berlin, Heidelberg, 200–219. https://doi.org/10.5555/648334.755721
[17] Simon L. Peyton Jones. 1987. *The Implementation of Functional Programming Languages.* Prentice Hall, Hemel Hempstead, Hertfordshire, England.
[18] Simon Thompson. 1991. *Constructive Type Theory and Functional Programming.* Addison-Wesley, Boston, MA, United States.