EOLANG Compiler.Stage 2

HSE Team
hsalekh@hse.ru
HSE
Moscow,Russia

Abstract

The EO programming language is an object-oriented language that is being developed as an RD solution, the purpose of which is to show that industrial programming in the pure OOP paradigm is possible. The language is based on the philosophy "Elegant Objects" and a fundamentally new formal model of -calculus, which defines basic operations on objects, positioned as necessary and sufficient to achieve object-oriented properties of the language.

Egor Bugaenko, the author of the "Elegant Objects" concept underlying the EO language, identifies the following problems of existing object-oriented programming languages:

1.Mutability of data generates side effects, which leads to unpredictable behavior of the program.

2.Using null references to denote the absence of a value.

3. Applying reflection breaks the encapsulation of classes, allowing the developer to use classes in unpredictable scenarios.

4. Inheritance allows you to get rid of visible duplication of code in similar classes, however, making any changes to parent classes only becomes more complicated because of the need for consistency of changes with all inherited classes.

Keywords: Elegant Objects, EOlang, Compiler, Java, OOP, Maven, Decoration, Datarization, Duck Typing

1 The aim

1.1 The aim

This work aims to develop and contribute an EOlang compiler that will be accessible at Github repository (OSS, MIT), which can build java classes from EO source code. Compiler Maven plugin wrapper to start compiler at build phase to generate class files, which will be packaged by Maven later. Contribution of such a solution will bring us to answer the question: Is it possible to rewrite Java Cloud web-module with EOlang.

2 Solution

The development of the compiler was devided in the separated parts that must cover the following features:

1) Translation of EO program code into ".java" files in the Java programming language.

2) Programming languages used in the implementation of the source code of the translator, objects of the standard library, as well as the test base (including languages used to supplement the existing code base) - Java, EO.

3) The parser (as well as the lexer) must be implemented (and augmented) as part of the ANTLR4 solution.

4) The preprocessing of the object structure after the parsing process must be done through Java transformations of XML documents.

5) The final solution (translator) should be delivered in an easy-to-use form: a Maven plugin in the Maven Central repository.

2.1 EO Objects Translation

This section describes how the translation of the user-defined EO object will be performed

2.1.1 Naming Scoping Rules. An EO object (say, named obj) of the global scope (0th level of nesting) is translated into a Java class named EOobj.java. (prefix EO is used as a simple measure to escape naming conflicts with Java code).EO 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 managing scoping and child-parent hierarchies to the Java Virtual Machine, not to the EO Runtime implementation. The source program obj object is translated into a single EOobj class put into a single EOobj.java file. No manual naming and class connections management. The transpiler is to compile the source program to the target platform just the way it is organized originally and let the Java Virtual Machine deal with all the relationships,hierarchies, and scoping mechanisms locally (native).

2.1.2 Target Class Hierarchy.

The target class extends org.eolang.core.EOObject.

2.1.3 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 of their appearance 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.

2.1.4 EO Objects Free Attributes Translation.

Free attributes of the source object are translated as follows. Say, there are these free attributes in the source EO object (in the order of appearance):

1. a
2. b
3. c
4. class
5. he133

For each of them, the following thing will be performed:

1. Generation of a private final class field named EOattr (EO prefix is used to allow using some special names in the source language like class, for example). The field is of type EOObject.

2. Generation of a public method named EOattr (EO prefix is used to allow using some special names in the source language like class, for example). 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 EOattr 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 regarding managing free attributes.

However, the model has some weaknesses. First, it cannot handle the partial application mechanism. The proposed solution allows the "fully-apply-at-once" copying technique only.

Secondly, just as the reference CQFN transpiler, the model does not perform type checking of the objects being passed for free attributes binding.

2.1.5 EO Objects Bound Attributes Translation.

Bound attributes of the source object are translated as follows.

1. For every bound attribute (named attr), a method named EOattr is generated (EO prefix is used to allow using some special names in the source language like class, for example). 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. The target code is generated as follows in this case:

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 their appearance 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 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 specified order in the source program if there are any of them.

d. The evaluation strategy is implemented via overriding the getData standard method (as described in detail in the Dataization section).

e. Access to all attributes inside the inner class are closed to the scope of the private class by default (so it is translated explicitly as “this.attr()” to make the target code determinable and comply with the explicit syntax rules of accessing the parent’s attributes in the source language).

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 explicitly translated to “EOobj.this.EOattr”.

It is important to mention that memory leakage issues that inner classes are blamed for do not affect the proposed model performance comparing it to the reference CQFN implementation 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 delegates managing scoping and child-parent hierarchies to the Java Virtual Machine, not to the EO Runtime implementation.

2.1.6 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 anonymous inner class of the EOObject class right in the context it is originated from. All the principles of class construction take place in the anonymous inner class just as in named classes (constructor generation, free and bound attributes translation, decoration dataization mechanism implementation, etc.).

2.1.7 Decoration Dataization.

As in the reference CQFN implementation, dataization in user-defined EO objects is highly connected to the decoration mechanism in the proposed model. Every user-defined object evaluation technique is implemented via an overridden getData 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

2.1.8 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.

2. Invoke method passing all the arguments into it in the order specified.

2.1.9 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 more simple way comparing to the guidelines of the proposed model. For example, bound attributes of the data primitives 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 with no base on the decoratee as it is guided above. Instead, these dataize to the atomic data or perform a more efficient (semi-eager or eager) evaluation strategy.

3 Deliverables

3.1 EO to Java Transpiling 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.Set a parent object 4.Access attributes of an object
EOData	Used to store data primitives
EODataObject	A wrapper class to interpret EOData as EOObject
EONoData	A primitive class representing the absence of data

3.2 Mapping EO Entities to Java Code Structures

EO	Java
Abstraction	A plain old Java class extending EOObject
Application	Creating a new class instance
A free attribute	A class field that should be set via the constructor
EONoData	A separate class with the following naming pattern EObase EOattr
Duck typing	Everything is EOObject. Class fields are accessed via Java Reflection
Object dataization	Calling getData()

3.3 Examples

3.3.1 Sample EO Program (covers almost all language features).

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

3.3.2 0-th level global object rectangle.

```
 <o line="5" name="rectangle" original-name="rectangle">
<o line="5" name="a"/>
 <o line="5" name="b"/>
 <o base=".mul" line="6" method="" name="area">
    <o base="a" line="6" ref="5"/>
    <o base="b" line="6" ref="5"/>
 </o>
 <o base="rectangle$perimeter"
    cut="5"
    line="7"
    name="perimeter"
    ref="7">
    <o as="a" base="a" level="1" ref="5"/>
    <o as="b" base="b" level="1" ref="5"/>
   <o as="area" base="area" level="1" ref="6"/>
 </o>
</o>
```

1. free attribute a

2. free attribute b
3. application-style bound attribute area
4. abstraction-style bound attribute perimeter

Another example is shown in the appendix section.

4 Conclusion

4.1 Performance Metrics Comparison

The solution being proposed is 17x faster and more efficient.

4.2 The Current (CQFN) Implementation

30th Fibonacci number is 832040 real 1m8,963s user 1m9,882s sys 0m1,391s

4.3 The Approach being Proposed

50th Fibonacci number is 12586269025 real 0m0,430s,user 0m0,652s sys 0m0,065s

A Appendix

A.1 Examples

A.1.1 findindex.eo.

```
    +package sandbox

[arr predicate] > findindex
  subFindIndex > @
    arr
    predicate
    0

[arr predicate i] > subFindIndex
  if. > @
    eq.
      arr.length
      i
    -1
    if.
      predicate.execute
        get.
          arr
          i
      i
      subFindIndex
        arr
        predicate
        add.
          i
          1
```

A.1.2 EOFindindex.java.

```
package sandbox;

import eo.org.eolang.core.EOObject;
import eo.org.eolang.core.EOObjectArray;
import eo.org.eolang.core.data.EOData;
import eo.org.eolang.core.data.EODataObject;

public class EOFindIndex extends EOObject {
    private EOObjectArray arr;
    private EOObject predicate;
    EODataObject i;

    public EOFindIndex(EOObjectArray arr, EOObject predicate){
        this.arr = arr;
        this.predicate = predicate._setParent(this);
        i = new EODataObject(0);
    }
    @Override
    public EOData _getData() {
      return _getAttribute("SubFindIndex" , arr, _getAttribute("predicate"), _getAttribute("i"))._setParent(this)
        ._getData();
    }
}
```

A.1.3 EOFindIndex$EOSubFindIndex.java.

```
package sandbox;

import eo.org.eolang.calc.EOadd;
import eo.org.eolang.calc.EOequal;
import eo.org.eolang.calc.EOif;
import eo.org.eolang.core.EOObject;
import eo.org.eolang.core.EOObjectArray;
import eo.org.eolang.core.data.EOData;
import eo.org.eolang.core.data.EODataObject;

public class EOFindIndex$EOSubFindIndex extends EOObject {
    private EOObjectArray arr;
    private EOObject predicate;
    private EOObject i;

    public EOFindIndex$EOSubFindIndex(EOObjectArray arr, EOObject predicate, EOObject i){
        this.arr = arr;
        this.predicate = predicate._setParent(this);
        this.i = i._setParent(this);
    }

    @Override
    public EOData _getData() {
        return new EOif(
                new EOequal(arr.length, i),
                new EODataObject(-1),
                new EODataObject(
                        new EOif(
                                new EOequal(arr.get(i), _getAttribute("predicate")),
                                _getAttribute("i"),
                                new EOFindIndex$EOSubFindIndex(arr, _getAttribute("predicate"),
                                new EOadd(_getAttribute("i"), new EODataObject(1)))
                        )._getData()
                )
        )._setParent(this)._getData();
    }
}
```