EO Objects Translation

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

Naming & Scoping Rules

An EO object (say, named **obj**) of the global scope (0th level of nesting) is translated into a Java class named EO**obj**.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 are originated from. So, attributes are named private inner classes, and anonymous objects are in-place 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 EO**obj** class put into a single EO**obj**.java file. No manual naming and class connections management. Let the transpiler compile the source program to the target platform just the way it is organized originally and let the Java Virtual Machine hold all the relationships, hierarchies, and scoping mechanisms natively.

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 of their appearance in the source program. All arguments of the constructor are of type **EOObject**, so the object (that we put the arguments into) does not know the actual type. In the case if free attributes are present, the default constructor is disabled (which is the built-in Java feature (if we have at least one constructor, the default one is not generated)).

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 EO**attr** (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 EO**attr** (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. As already have been mentioned in the Constructors section, the generation of an argument in the sole constructor (in the order of appearance of the free attribute in the source object) takes place as well. 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 access one and the only constructor with all free attributes passed to it. Fields holding free attributes are private and final. The default (empty) constructor is disabled. Once sat, 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. Second, just as the reference CQFN transpiler, the model does not perform type checking of the objects being passed for free attributes binding.

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 EO**attr** 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 EO**attr** 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 EO**attr** 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 EO**attr** that wraps the EO**attr** private inner class returns a new instance of that class passing all arguments (free attributes) to it in the order of their appearance 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. All attributes access inside the inner class are enclosed to the own scope of the private class by default (so it is translated explicitly as **"this.attr()"** to make the target code determined 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 (say, EO**attr**) of the parent object (say, EO**obj**) of the attribute (say, **EO**boundAttr), 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 are originated 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.

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

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.

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 EO**attr** 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 of their appearance.

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.