Автономная некоммерческая организация высшего образования
«Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)
по направлению подготовки
09.03.01 - «Информатика и вычислительная техника»

GRADUATION THESIS
(BACHELOR'S GRADUATION THESIS)
Field of Study
09.03.01 – «Computer Science»

Направленность (профиль) образовательной программы
«Информатика и вычислительная техника»
Area of Specialization / Academic Program Title:
«Computer Science»

Тема / Topic	**Проецирование Java в φ-исчисление и его реализацию — EOLANG / Mapping Java to φ-calculus and its implementation — EOLANG**

Работу выполнил / Thesis is executed by	**Степанов Максим Александрович / Maxim Stepanov**	подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis	**Зуев Евгений Александрович / Evgenii Zouev**	подпись / signature

Иннополис, Innopolis, 2022

Contents

List of Tables

List of Figures

Abstract

This thesis describes a way to map a traditional Object-Oriented Programming Language (namely — Java) to phi-calculus (mainly to its implementation — EOLANG). Java is one of the most popular object-oriented languages and is heavily used in the industry. Phi-calculus is a formal model for the representation of object-oriented languages. The main result of the project is a translation tool for the conversion of Java source code into EOLANG source code.

Chapter 1

Introduction

This project aims to provide a transpilation tool for further analysis of Java source code (*source language*) by translating it to EOLANG (*target language*) and passing the result to Polystat **polystat_repo** static analyzer.

EOLANG **eolang_repo** is a universal intermediate representation for object-oriented language semantics actively developed under the Polystat project. EOLANG, and its formal base — φ-calculus — are not thoroughly researched yet, which gives **novelty** to the project and is the main reason for its choice as a target language for the project.

Java is the third-popular programming language according to the TIOBE index **tiobe_index**. TIOBE Index is a respected metric in the field of software development. The popularity of the language gives **actuality** to the project, which is the main reason for its choice as a source language for the project.

Static semantic analysis **bardas2010static**, **epure2016semantic** is one of the popular approaches to minimizing the number of faults in a software project, and the field of static analysis is actively researched nowadays.

The overall goal of Polystat is to provide tools for the translation of vari-

ous programming languages into EOLANG (in which category the current project falls) and a tool to perform generalized static analysis on the EOLANG intermediate representation. This approach allows implementing a single universal analysis tool to cover several object-oriented languages, minimizing the effort of the Polystat analyzer development team by only implementing the analysis algorithm once. The structure of the entire project is shown in figure 1.1.

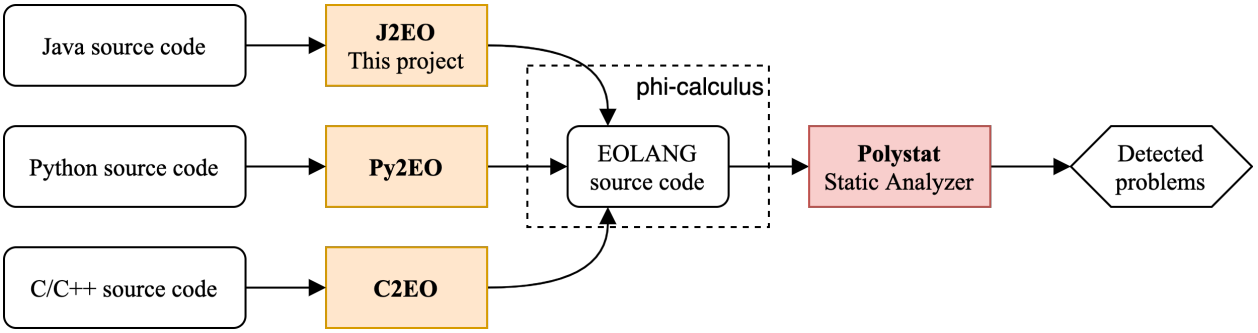

Figure 1.1: Overall structure of Polystat project

Literature Review chapter will outline the general information about programming language translators, work done in the field, goals of such projects, problems commonly faced and their solutions. It also covers some of EOLANG's features and problems associated with them.

Java Overview chapter will describe source language structure and most notable features which matter for the project.

EO Overview chapter will outline target language structure and most notable features which matter for the project.

Conceptual comparison will outline the differences between source and target languages and state most problematic cases for translation between the two.

Implementation chapter will provide projecting methodology in depth, as well as describe the project implementation details.

Results and Discussion chapter will present the outcome of the implemented

project and discuss the future possibilities and actuality of the project.

Chapter 2

Literature Review

2.1 Basic information

From the beginning of programming history, developers wanted to improve available tooling and automatize time-consuming and error-prone operations. One of such tasks is moving a code already implemented in a programming language A to language B, benefitting "software reuse". For that task, a source code translator is used.

Some translation tools (and their appropriate languages) improve a non-ideal language by packing additional features — e.g., adding generics to Java back in days when the language did not support them **java_genericity**. The popular extended language that affects many modern developers is JavaScript, which is extended into TypeScript, CoffeeScript, etc. **unsupervised_translation**

The other type of tool is meant for the translation of two heavily different (in approaches or even paradigms) languages into each other. This particular direction is the hardest, as different languages use different underlying APIs, forcing developers of a translator to either map calls to the language's standard library, or

write a wrapper for it. Three main approaches used for that are hand-crafted mappings of constructions from language A to language B, manually-written wrappers, and training a neural network on a dataset of parallel source code snippets, as did Marie-Anne Lachaux et al. **unsupervised_translation**.

The overall goal of this project is, however, not to produce a fully equivalent code, with or without manual intervention. The goal is to map as much of Java semantics as possible to allow further static analysis of generated code to find bugs. EOLANG was chosen for the task as it is an implementation of φ-calculus **eolang_phi_calculus**, a mathematical foundation for "true" Object-Oriented Programming. "True" in this context means that everything is an object: data, classes, class instances, and even methods. And since φ-calculus is a purely mathematical model, it is possible to rely on mathematical theory (in particular, set and graph theories) developed for centuries and use its theorems.

2.2 Standard Library

As for the work done directly in relation to EOLANG: since the runtime for this language is written in Java, there is a possibility to use Java standard library directly without translating it, using interoperability. This technique is broadly discussed in **design_evaluation_interoperable_translation_system**. This significantly reduces the difficulty of bringing a standard library to another language. A group of researchers has followed exactly that path: they implemented an EOLANG wrapper that is able to call Java functions from EOLANG **exploring_eolang_integration_interoperability**. This approach still requires us to manually implement all of the Java standard library classes and fill them with wrappers that call original Java functions, though. An approach to handling such

classes is discussed later in the literature review.

Difficulties similar to this project were researched by Pablo Arrighi et. al **the_gool_system**. The authors mentioned major differences in languages being translated and the need of replacing language-specific constructs and traits with more generic ones. Also, the problem with mature old languages like Java (which we are translating from in this project) is a large amount of syntactic sugar — constructions that may be represented by the core language features, but are added for users' convenience. However, while simplifying the life of language users, this adds complexity to the tools that work with the language source code: translators and processors.

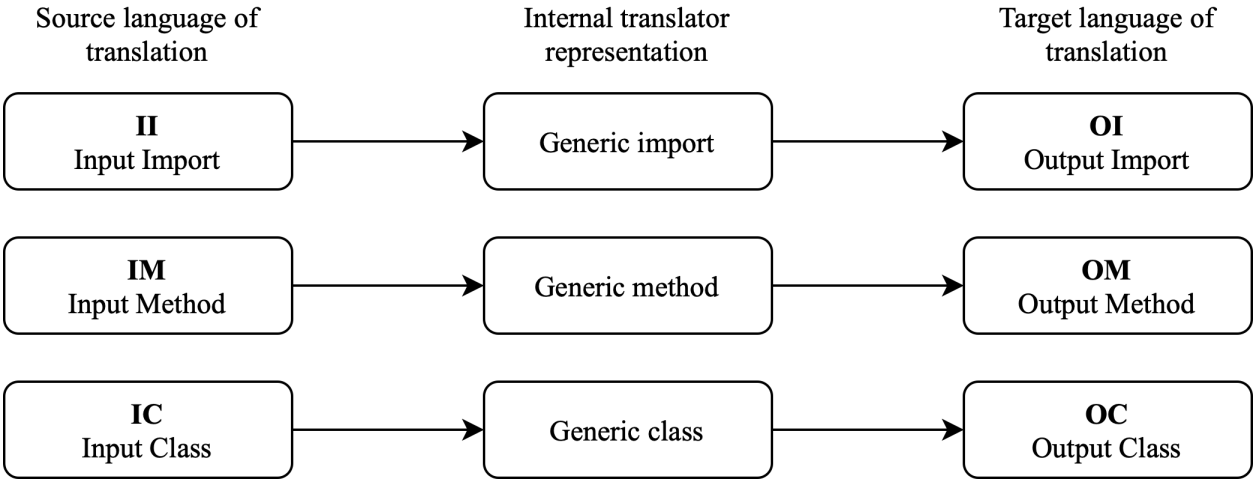

Figure 2.1: Usage of generic abstract definitions for universal mapping as per Pablo Arrighi et. al

To solve the standard library difference issue the same authors proposed an elegant solution of adding generic abstract definitions in the intermediate language that are mapped to each language's library functions. They consist of three construction types in the direction of original language to generic language, as shown in figure 2.1: II — import mappings, IC — class mappings, and IM — method mappings, and corresponding OI, OC, and OM mappings in the direction of generic

language to the specific language. Since the translator implemented in our paper only does one-directional mapping, the second part may be dropped. And the generic representation is, in this case, replaced with EOLANG. This may be combined with the wrapper mentioned above to produce a controllable yet automatized approach to mapping the standard library.

2.3 Testing

To carefully assess the result of a translator, automatic testing is needed. It may take the form of parallel functions, an idea mentioned in **unsupervised_translation**. Despite it being used as a training dataset for an artificial neural network there, one of our testing framework approaches is comparing the output of the translator to a predefined, manually written parallel source code file for equivalence. The other is executing the original Java snippet, reading its output, executing produced EOLANG snippet, and comparing the stdout results for equivalence.

2.4 Immutability of data in EO

To better understand this section, you might read the chapter named ”EO Overview” 4.

One of EO’s basic principles is the immutability of data, as per the description and original paper **eolang_phi_calculus**. This is the approach followed mostly by functional languages, not imperative languages, although, there are exceptions **coblenz2016exploring**. Mainstream OOP languages are imperative and consequently have data mutability at their base. Translating a language with mutable

data to a language with immutable data poses a considerable challenge.

The closest operation to *assignment* in conventional languages is *binding* in EO, though, it is not semantically equivalent. The resemblance is only logical. While assignment mutates data in place, binding connects several references (attributes in EO) using a given operation or function. ”Several” includes output attribute and all attributes participating in the expression to bind. As discussed later, objects in EO are represented as graph nodes internally. Binding creates an edge between two nodes located on the left-hand side and right-hand side. Binding creation happens at compile time. As the result, once an attribute is bound, it cannot be rebound again (i.e., the following mapping of a snippet in Java to a snippet in EO is invalid):

```
1 void function() {
2     int i = 1;
3     i = i + 2;
4 }
```

```
5 [] > function
6   1 > i
7   i + 2 > i
```

One approach to solving the problem of mapping mutable operations to immutable operations is the usage of Static Single Assignment (SSA) **ssa_1**. Even though it is usually used in compilers to perform optimization of programs, the main idea of SSA is to transform each mutation operation (like assignment, increment and others) to assignment to a new variable. This is perfectly suitable for EO’s binding operation, as the variable is not reassigned. Example of SSA mapping:

```
8   [] > function
9     1 > i_1
10    i_1 + 2 > i_2
```

However, due to the specifics of SSA, loop variables are still updated in place. This can be mitigated by replacing loops with a recursive call to a loop body, which is possible to perform in a general case.

One problem still persists: there are global variables in mainstream object-oriented languages, which cannot be transformed to SSA in the general case, as this would lead to carrying a global state in each operation.

This problem expands the boundary of the project significantly, thus, I decided to exclude it from the scope of the thesis and project. As for the current solution, mappings fall back on the usage of built-in EO atoms for the simulation of mutable data attributes. It is planned to solve this problem later, in a separate project. This thesis does not cover the mapping of mutable variables to immutable attributes.

Chapter 3

Java Overview

The document and project use UNIX-style notations for file paths since UNIX systems are most popular among developers, who are the target audience of both this thesis and the project.

The incoming data of the project is Java source code, thus, the separate section is dedicated to outlining the syntax and semantics of the language. In-depth description of Java specification is available on the Oracle website **java_specification**.

3.1 Project structure

Typically, Java projects consist of:

- Build system configuration files; two major ones are Maven **maven_repo** and Gradle **gradle_repo**. Project dependencies are declared in these files as well;
- `src` directory with source code;
- Markdown files, including README file, wiki, documentation, etc.;

- Optionally, configuration for CI/CD and other tools;

Complex projects may break up their structure into modules, which are essentially mini-projects isolated under a subdirectory with the file structure specified above.

Depending on the configuration of build system, source code may be split into different source sets **gradle_sourcesets**, commonly `main` and `test`. In this case, `src` directory includes subdirectories for each source set which act as a root for packages. In the case of a single source set for a project, `src` acts as a root of the source set. *From now on, until the end of this chapter, the text assumes that paths start from the source set directory*.

The translated EO code is passed to the static analyzer which detects problems in the current project alone. Thus, dependencies should not be included in the resulting code in any way. This allows the project to ignore the configuration of the build system and only process `src` directory.

3.1.1 Source code directory structure

Java projects have a strictly defined source directory structure **java_specification**: nested directories correspond to the package name specified in the source file, and the file name corresponds to the class name specified inside the file. For example, file `Main.java` located with a specified package `org.polystat.j2eo` should be placed strictly into `org/polystat/j2eo/Main.java`.

3.1.2 Language entities visibility

Java provides four levels of visibility for all the entities:

- `public` — available from any class from any package;
- `protected` — available from any class that inherits from the class that contains the entity;
- `private` — available from the current class only;
- package-private — available from any class from the current package. When no visibility modifier is specified, package-private is used by default.

3.2 Source code file structure

Java source code file has a strict structure, which includes:

- Package declaration — optional for a single-file project, but required for multi-file projects;
- imports — optional;
- class or interface declarations. Strictly one public class should be present in the file, so importing it deterministically resolves the imported class.

Class structure

Java classes may consist of:

- static members (variables);
- static methods;
- non-static members;
- non-static methods;

- nested classes/interfaces.

Member declaration

Member declaration may include:

- visibility modifier 3.1.2;
- static modifier;
- final modifier;
- other modifiers not covered by this project (e.g. volatile, transient);
- (required) type of the member, or `var` for automatic derivation of the type based on usage;
- (required) name of the member;
- value of the member;

Method declaration

Method declaration may include:

- visibility modifier 3.1.2;
- static modifier;
- (required) return type of the method;
- (required) body of the method;

3.2.1 Method body structure

Method body includes any amount of statements.
Statement may be:

- assert — if condition specified inside is false, throws exception;
- if-then-else — classical conditional block;
- for loop — loop with pre-action, condition and post-step action;
- while loop — loop with condition only;
- do-while loop — loop with condition that executes at least once;
- try-catch — executes block and gracefully handles exceptions thrown inside;
- try-with-resources — gracefully handles disposable resource and executes a block;
- switch — sequentially checks the given expression for specified conditions;
- synchronized — uses given mutex to avoid multi-threading problems and executes a block;
- return — exits the function optionally returning a given value (if function return type is not `void`);
- throw — throws the given exception and aborts execution to the point of the nearest handling catch clause (or crashes the program if there is none);
- break — aborts the nearest loop;

- continue — stops processing of the current loop step and goes into the beginning of a loop;
- expression — evaluates an expression, but does not assign its result to a variable;

Example of a class including above declaration descriptions:

```
11 package org.polystat.j2eo
12
13 import org.polystat.j2eo.StaticData
14
15 public class Main {
16     private final dataToPrint = StaticData.helloWorldText;
17
18     public static void main(String[] args) {
19         System.out.println(dataToPrint);
20     }
21 }
```

3.2.2 Expression structure

TODO: add detailed expression description

3.2.3 Interface structure

The original meaning of interfaces in Java was to split declaration and definition of entity, facilitating Encapsulation/Data Hiding.

Though, in recent versions of the language interface feature list has grown, adding *default* methods. This allowed interfaces to have the definition alongside the declaration, essentially allowing the implementation of multiple inheritance in Java.

In object-oriented languages, this unavoidably leads to the diamond problem **sakkinen1989disciplined** — situation when several inherited interfaces of a class contain a default method with the same name. Different languages solve this problem in different ways. Java compiler aborts compilation whenever it detects such a case.

Chapter 4

EO Overview

EO is an object-oriented language created to act as a universal intermediate representation for static analysis of object-oriented languages. Universal here means that it supports different implementations of the same concept in different languages.

Static analysis requires a solid foundation beneath, so EO uses phi-calculus as its own foundation. Phi-calculus will be described after covering practical aspects of its usage in EO in section 4.3.

EO focuses on pure objects and a concept of decoration. It does not provide an implementation of classes or functions on its own; this would make the implementation non-universal, as languages implement these concepts in different ways. Instead, it provides capabilities for the developer to implement functions and classes on their own.

EO source code is the main product of this project, thus, the text includes a description of program structure, syntax, and features to facilitate understanding of the methodology, implementation, and results.

4.1 Source code file structure

Every EO file starts with meta-information, which includes:

- package — the namespace where all declared objects are located. Other files use the package to refer to objects located in the file. This construct has a form of `+package <package name>`.
- aliases — fully-qualified names (FQN for short) (package plus object name) used to import other definitions to the current file. This construct has a form of `+alias <FQN of object>`.

Example of meta-information block:

```
22 +package main
23 +alias org.eolang.txt.sprintf
```

Then declarations follow, which are the main construction blocks of EO programs.

Literal values syntax is built into EO, and contains:

- `"str"` for strings
- `42` for integers
- `4.2` for floating-point numbers
- `'c'` for single characters
- `AB-42` for bytes sequences (hexadecimal representation of bytes delimited by dashes).

Booleans are implemented in the standard library, they are objects.

Objects have the following syntax:

```
[free_attr_1 free_attr_2, optional_var_attr...] >
↪ object_name
    <expr> > bound_attr_1
    <expr> > bound_attr_2
    seq > @
        <imperative operation 1>
        <imperative operation 2>
        <result expr>
```

where:

- free attributes, marked as `<free_attr>`, are similar to arguments in traditional languages.
- bound attributes, marked as `<bound_attr>`, are objects holding an expression 2.4. They are logically similar to assignment in conventional OOP languages. Though, they can not be reassigned and do not evaluate until their result is *dataized*, e.g., computed, transformed from a declaration to a particular value by one of the imperative operations, such as print.
- expressions, marked as `<expr>`, take a form of:

```
number1.add
    number2
```

or

```
add.
    number1
    number2
```

both of which are semantically equivalent.

Like with attributes, to which they can be bound, expressions are not evaluated until the result is dataized.

Object declaration is an object as well, which means such constructions can be nested.

- `... > @` marks the *decoration* operation: in a nutshell, it takes the referenced object (marked as `...`) and wraps it with additional attributes specified in the object. This provides an approach to implementing the singular inheritance of objects.

  For example, in the following snippet, object `B`, which decorates `A`, will contain both `first` and `second` as its bound attributes:

```
36 [] > A
37   1 > first
38
39 [] > B
40   A > @
41   2 > second
```

  The meaning of `seq` will be discussed later.

These are all EO built-in mechanisms. It can be seen that EO does not have built-in classes. The implementation of the class framework will be described later in the text.

The powerful part of EO lies in its atoms. It is not possible to write a program in EO without atoms because dataization can only happen there. And without dataization, the program is a bare declaration, not differing semantically from YAML or JSON.

Atoms include both functions (i.e., print) and data objects, most notable of them are:

- booleans,

- `memory` object that allows to store mutable primitive values, including strings, numbers, characters and bytes,
- `cage` object that allows to store objects,
- `seq` function that allows sequential computation.

One particular atom that is very important to this project is `seq`: it provides a way to express imperative computation in EO. It works by accepting an arbitrary number of expressions as arguments. All except the last one are dataized to actually perform the computation. The last expression is returned as-is because dataized expressions yield a primitive value, which does not have the original structure, original nested attributes and are practically useless for further operations. The final dataization usually happens in `print` operation or other atoms performing side effects.

`seq` allows objects to simulate function behavior from traditional languages. In this text, I will refer to objects with `seq` block as **functions**. While this is not entirely correct syntactically, this will ease the reading of the text, as **function** is a widely-used term for such construction.

4.2 Analysis of EO suitability for the goal of this project

EO focuses on pure objects, but the term ”pure” may be extended to OOP languages as well. With respect to object-oriented languages, this term means that the following conditions hold **making_pure_oop_languages_practical**:

- (1) Language provides Encapsulation/Data Hiding;

- (2) Language provides Inheritance;
- (3) Language provides Polymorphism;
- (4) Language provides Abstraction;
- (5) All predefined types in the language are objects;
- (6) All user-defined types in the language are objects;
- (7) All operations performed on objects must be only through methods exposed at the objects.

Having a pure language facilitates richer static analysis, as the amount of uncontrolled side effects **gifford1986integrating** is minimized, and they can be analyzed. The main goal of the project is to provide the most suitable representation for further static analysis, thus, checking these properties is essential.

4.2.1 How do these seven properties ease static analysis?

First, it is important to check how these properties help to improve the quality of static analysis and is it worth to fulfill them.

- Encapsulation allows skipping the check of external direct data access, as it hides the data behind functions. As such, data access should only be checked inside the class declaration.
- Inheritance, polymorphism, and abstraction do not help with analysis, they are designed to help developers.

- Defining built-in types as language objects helps by minimizing the analyzer's built-in logic, as this logic may be deduced from the source code of the standard library.
- Defining all user types as objects helps by removing handling of custom type logic; all analysis focuses on objects solely.
- Shrinking range of object manipulation to only methods exposed by the object itself plays the same role as encapsulation: analyzer does not have to consider external object manipulation other than the logic defined in the class itself.

Four of seven properties make sense to fulfill to reach the goal of the project, so I will focus on them.

4.2.2 Checking EO against pure OOP language properties

Encapsulation/Data Hiding

EOLANG paper **eolang_phi_calculus** states that the language does not support encapsulation/data hiding, this property is missing.

Predefined types are objects

Objects are the main focus of the EO language, there are only objects. Literals are represented as objects as well. This property is fulfilled.

User-defined types are objects

Same as with predefined types, there is no way to define non-object types.

Methods defined on the object are the only ways of object manipulation

Since there is no assignment in EO and the only type is the object, the only way to interact with objects is to call its methods.

Conclusion

Three of the defined properties hold for EO. Thus, it makes sense to fulfill them in the project.

4.3 Phi-calculus overview

φ-calculus **eolang_phi_calculus** is based on lambda-calculus, formulated by A. Church **church2001logic** and set theory. The simplified description of underlying calculus is given to facilitate a better understanding of EOLANG hard boundaries.

4.3.1 Objects

The core entity of φ-calculus is an object. Object *o* is a set of ordered pairs named *attributes* such that keys *a* are identifiers and values *v* are objects. An identifier may be one of the reserved meaningful values: φ, ρ, σ, ν, or any text without spaces starting with a lower-case English letter:

$$o = \{(a_1, v_1), (a_2, v_2), ..., (a_i, v_i)\}, \text{ where } i \in [0, \infty)$$

Empty objects are allowed. An empty object is denoted by $\varnothing$, as in classic set theory.

The *scope* of an object $o = \{(a_1, v_1), ..., (a_i, v_i)\}$ is denoted with $\hat{o}$ and is a set of $a_1, ..., a_i$.

The *arity* of an object o is the cardinality of its scope and is represented as $\mid \hat{o} \mid$.

From the description of the object it is clear that φ-calculus was designed by developer for developers — multi-character identifiers are first-class units, unlike in mathematics. Thus, treating it as a programming language notation instead of mathematical notation may help to understand this calculus better.

4.3.2 Data

The indivisible unit of φ-calculus is *data* — the possible values of data depend on the implementation platform, but generally-available types include integer numbers, floating-point numbers, strings, bytes, byte sequences and booleans. Data is an object as well. For the purposes of this project, available data types are the ones available in the EOLANG compiler. For the list, refer to chapter 4.

4.3.3 Attributes

Attributes are divided into two types:

- *free attributes* — attributes which have $\varnothing$ as a value. Formally, attribute a of object o is free iff $(a, \varnothing) \in o$.
- *bound attributes* — attributes which have non-empty set as a value. Formally, attribute a of object o is bound iff $\exists(a, v) \in o$ where $v \notin o$.

Accessing attributes

If object o has an attribute named a that contains value v (formally — $\exists(a,v) \in o$, for example, $o = \{(a,v)\}$), then v may be accessed using dot notation: $o.a$.

Dot notation may be chained if v is an object as well. v in the following example may be accessed with $o.a.b$:

$$o = \{(a, \{(b,v)\})\}$$

Both bound and free attributes may be accessed this way.

4.3.4 Abstraction

If an object contains at least one free attribute, it is named *abstract object*. The process of creation of an abstract object is called *abstraction*.

If an object contains no free attributes, it is named *closed object*.

4.3.5 Application

Application is an operation defined on abstract objects that creates a new object with one or more of free attributes becoming bound. Formally, if x is an abstract object and y is object with $\hat{y} \subseteq \hat{x}$, then the resulting object o will contain pairs with values as follows:

$$\left(a \in \hat{x}, v = \begin{cases} x.a \text{ if } x.a \neq \varnothing \\ y.a \text{ if } x.a = \varnothing \end{cases} \right)$$

If not all attributes become bound after application, the newly created object is abstract, otherwise — the newly created object is closed.

4.3.6 Parent and home objects

J2EO does not use parent and home objects, but they are still present in this description since they may facilitate new feature implementation in the future.

For object o, $o.\rho$ represents its *parent* object, i.e., the object that created o.

For object o, $o.\sigma$ represents its *home* object, i.e., the parent object of abstract object from which o was created.

4.3.7 Decoration

Decoration is the core operation of φ-calculus. *Decoratee* is denoted with φ; this symbol gives the name to φ-calculus. *Decoration* is the operation which extends object o with additional attributes to produce new object x. This operation is similar to inheritance in object-oriented programming languages.

4.3.8 Atoms

Atoms are defined by the runtime, and thus, described in the EOLANG description — the runtime used by J2EO.

4.3.9 Locators

φ-calculus is used to represent source code, which often spans across several files and may contain thousands of object declarations, some of which are named with the same identifier. To deterministically resolve such conflicts, φ-calculus

contains *locators* — sequence of identifiers separated with dot. Φ denotes a root object — all top-level object declarations are bound to it.

4.3.10 Identity

φ-calculus is used to represent abstract entities, and as such, objects should contain built-in identity. For object o, $o.v$ is a positive integer data object that is unique to o in the entire runtime scope. There's no guarantee that this number is always the same across different environments and executions, but it strictly stays the same after the object is created.

4.3.11 Summary

φ-calculus covers types, objects, and operations on them, providing all facilities to implement the general semantics of Java programs.

4.4 Internal representation of EOLANG

EOLANG is essentially a phi-calculus wrapped into a programming language, with a convenient syntax for writing programs on a standard computer keyboard. It also includes features that make it useful for general-purpose tasks, such as the I/O library, math library, built-in data structures, etc.

EO compiler outputs Java source code which creates a phi-calculus graph that describes the program. This fact affects the boundaries of allowed modifications to EOLANG, as we can't implement features that do not correspond to phi-calculus.

Chapter 5

Conceptual comparison of Java and EO

5.1 Visibility

Java has a strict visibility system that includes four levels described in subsection 3.1.2. EO, on the other hand, has no visibility system at all. Such visibility system is hardly possible in EO given its dynamic nature, including dynamic typing, lack of object structure specification and unlimited access to the creation and modification of objects. These features are required to keep the language structure flexible enough to translate various languages into it.

5.2 Mutability

In Java, all variables are mutable by default (though it may be limited using final). EO has no built-in mutability, but it has workarounds for that in the standard library: `memory` and `cage`.

5.3 Coding style

Java has an imperative style — developer writes a code that describes *how* to do the task. Declarative style may be achieved using custom-implemented pair of executor and data structure, but it will not work soundly with libraries (both standard library and external libraries). Custom wrappers are needed to support libraries.

EO has a declarative style — developer writes *what they want to achieve*, not how to achieve that. Imperative style may be achieved using a built-in atom named `seq`, but because of the implementation nature, the scope of `seq` is more restrictive than the object scope, e.g., it is not possible to declare variables inside.

TODO: add something else?

Chapter 6

Implementation

6.1 Projections

This section will go over theoretical approaches of projecting Java source code into EO source code.

6.1.1 Naming

Java and EO have different conventions for naming language components, and while Java does not enforce naming conventions, EO does. Consequently, name mapping has to be developed and be consistent across the translator to keep the result code working.

Specifically: object and attribute names in EO can only start with lower-case letters and continue with upper/lower case letters, numbers, dashes or underscores. This differs from Java only in the first symbol, therefore, the solution is to prepend names with fixed text sequence.

Also, EO has problems with attribute shadowing. Attribute shadowing is a feature of a language that allows redefining variables (attributes in EO terms) in

nested scopes. This feature works in an unpredictable manner, especially considering the unstable nature of EO (it is still in the early stage of development and behavior changes frequently as of the time of writing this). Because of that, the translator avoids name duplication as much as possible. The first step to minimize such duplications is to prepend different types of source code tokens with different prefixes.

As a solution to the attribute shadowing problem, the following mappings were developed:

- class names are prepended with `class__`
- variables are prepended with `var__`
- arguments are prepended with `arg__`
- methods are prepended with `method__`

6.1.2 Constructors

Java, like many modern OOP languages, has class constructors. EO does not have classes, and thus the implementation of constructors is under full control of the translator. In the current form, Java constructors are mapped into function attributes with an extra `cons` name.

6.1.3 Overloading

Also, EO does not support method/constructor overloading and even does not have functions directly. They are simple objects, like everything in the language. Therefore, the name of methods/constructors are appended with type

names, separated with __. Generics information is omitted, as JVM does not have it as well.

Example of resulting mapping considering everything above:

```
class A {
  A(int member) {...}

  void functionName(int arg1, Option<Value> arg2) {...}
}
```

```
[] > class__A
  [] > new
    [] > this
      [var__member] > cons__int
        ...

      [] > method__functionName__int__Option
        ...
```

6.1.4 Expressions

Java provides a way to create complex nested and chained (separated with dot) expressions. These expressions can cause various side-effects in various order.

Chained expressions

The tricky case is chained expressions. Due to EO's specific function calling approach, calling a method of a class instance requires passing that instance as the first argument, i.e. `obj.function(obj)`. This is effortless when single-expression statements are used, but chaining with a dot requires storing the intermediate object somewhere to be able to proceed in the chain.

Consider the following Java example:

```
55  var result = obj.setValue(value).computeResult();
```

Here, `obj` should be passed between chained calls. To resolve this collision, it was decided to split each chain operation into a separate binding, and use them inside other bindings. This allows to abstract out the double use of the same value from higher-level abstraction.

Since internal operations do not act as dependencies to the outside of expression, their names are generated randomly and have a UUID format.

The order goes from inside to outside, so dependencies of outer expression parts are already defined above, although it does not matter to EO, as an evaluation only starts in the `seq` block at the end of the function object.

The example of EO code that corresponds to the above Java snippet:

```
56  ... > obj
57  ... > value
58  ...
```

```
[obj] > <uuid_1>
  seq > @
    obj.setValue obj value
[obj] > <uuid_2>
  seq > @
    obj.computeResult obj
...
seq > @
  ...
  <uuid_2>
    <uuid_1>
      obj
```

Although the EO variant takes considerably more code to do the same operation, the language was never meant for hand-writing code, only for translation from other languages.

Side-effects in expressions

Consider the following Java snippet:

```
var out = obj.setValue(obj2.setValue(obj2.getCount()).i++);
```

It is different from the previous snippet by one feature — post-increment operation, which is an in-place mutating side-effect.

In this code, in given order, the following operations occur:

- Function `getCount` of `obj2` is invoked and result is obtained
- Function `setValue` of `obj2` is invoked with the obtained value
- `obj2` returns itself for further chained call
- Property `i` of `obj2` is obtained

- Property `i` pf `obj2` is incremented by 1
- Function `setValue` of `obj` is invoked with value obtained before increment, as this is value type, not reference type
- The result of `setValue` is assigned to `out`

As it is seen from the list above, the order of execution does not correspond to the AST, in particular — post-increment operation. This operation should have been placed before obtaining the result, as in pre-increment, to match the AST structure. This fact of altered order makes Java code hard to translate to EO, which does not have corresponding syntactic sugar, as well as value-typed variables.

The general problem is causing side-effects from the expression: EO discourages any side-effect operations and only provides ”gray” workaround atoms to mimic side-effects, to create a compatibility layer with side-effect-based programs. But since there are still no value types, we have to work around operations that rely on them with custom logic (i.e., creating a new variable).

Considering methods described above, we can obtain the following EO mapping:

```
72  ... > obj
73  ... > obj2
74  ...
75  [obj] > <uuid_1>
76    memory > result
77    seq > @
78      result.write
79        obj.getCount obj
80      result
81  [obj arg] > <uuid_2>
82    seq > @
83      obj.setValue obj arg
```

```
84  [obj] > <uuid_3>
85    memory > result
86    seq > @
87      result.write
88        obj.i
89      obj.i.write
90        add
91          obj.i
92          1
93      result
94  [obj arg] > <uuid_4>
95    seq > @
96      obj.setValue
97        obj
98        arg
99  ...
100 seq > @
101   ...
102   <uuid_4>
103     obj
104     <uuid_3>
105       <uuid_2>
106         obj2
107         <uuid_1>
108           obj2
109   ...
```

Notice how post-increment was implemented in `<uuid_3>`: new `memory` object (think of it as of cell that can store some value) is created, the current value is copied there, and then the original object's value is updated.

The same approach is used everywhere where value types are used (for Java, the list includes `int`, `long`, `short`, `char`, `float`, `double`). In this case, it is used in `<uuid_1>`. This provides a reliable way to ensure that the original value will

not be changed by subsequent function calls, as it is not changed in Java.

6.1.5 Resolving compound names in Java

Java has a very unfortunate feature of referencing nested classes/members: every token is separated with a dot in any case. Therefore, it is impossible to tell which exact operation is represented by a particular dot: is it a part of a package name, referencing a class from a package or referencing a member of a class.

To determine the operation of the compound name, the import block is analyzed: if the line from the Java import block (excluding the last part, either `*` or class name) is a start of the compound name, parts of the package are not altered in the EO compound name; the file referenced by the package name is loaded, parsed and analyzed. parts that correspond to class names are prepended with `class__`, parts that correspond to variable names are prepended with `var__`.

6.1.6 Mapping classes

Class bodies are fairly straightforward to translate.

- class is an object with all static members and static methods, plus `new` object factory and `cons<n>` constructors. It also decorates the superclass, inheriting its static members and methods.
- `new` method decorates superclass'es `new` object and defines class members with their default values over that.
- `cons<n>` method that contains corresponding Java constructor code.

What poses a challenge is an inheritance, especially multiple inheritance (implemented via interfaces in Java). There are still no observed approaches to map

interfaces in the project, so they will be skipped for the time being. The focus will be on mapping classes.

Superclass is written to `super` attribute of an instance during its creation.

Example of class mapping:

Java code:

```
110  class A {
111    int i = 42;
112
113    int getI() {
114      return i;
115    }
116  }
117
118  class B extends A {
119    int i = 1;
120
121    B() {
122      super();
123      i = 2;
124    }
125
126    int getI() {
127      return i;
128    }
129
130    int getSuperI() {
131      return super.i;
132    }
133
134    int setSuperI(int i) {
135      super.i = i;
136    }
137  }
```

EO code:

```
  [] > class__A
  [] > new
    [] > this
      memory > i

      [this] > getI
        this.i > @

    seq > @
      this.i.write 42
      this

  [this] > cons1
    this > @

[] > class__B
  [] > new
    [] > this
      class__A.new > super
      super > @
      memory > i

      [this] > getI
        this.i > @

      [this] > getSuperI
        this.super.i > @

      [this i] > setSuperI
        seq > @
          this.super.i.write i
          this

    seq > @
```

```
172       this.i.write 1
173       this
174
175   [this] > cons1
176     class__A.cons1 this > this2
177     seq > @
178       this2.i.write 2
179       this2
```

Chapter 7

Results and Discussion

7.1 Results

The result of this thesis is a part of the software program J2EO written in Java/Kotlin. The implemented part includes EO Abstract Syntax Tree (AST), an algorithm that prints EO AST to the source text file, and an algorithm that translates selected Java AST nodes to EO AST. The source code may be found at GitHub **j2eo_repo**.

The main use case for J2EO is to translate real-world Java projects to EO to later perform static analysis with other Polystat projects. Thus, it is important to translate large projects without failures and maximize the amount of translated Java features. Producing full semantically-equivalent code is not a requirement for static analysis.

As the benchmark for result assessment, I have picked several big projects heavily used in production systems.

The table of used projects with their corresponding actual commit hashes is listed is table I:

Project	Actual commit hash
Hadoop **hadoop_repo**	ec0ff1dc04b2ced199d71543a8260e9225d9e014
Kafka **kafka_repo**	f36de0744b915335de6b636e6bd6b5f1276f34f6

Table I: Java projects used for benchmarking

What would be good metrics for the project?

I will populate this with actual data after our team will fix performance issues with expression mapping. For now, it takes forever to process either of the picked projects.

7.2 Discussion

The J2EO project already covers a significant range of Java features, facilitating the future development of the Polystat analyzer. Since Polystat heavily depends on the output of transpilers, its development was stalled for a long period and thus no rich analysis results are produced as of the time of writing.

J2EO/Polystat combination is still in active development and not production-ready. But according to the results of J2EO, EO provides a necessary base for the representation of other languages. This combination has a solid chance to become competitive with other popular Java static analyzers, such as PMD, SpotBugs, and IntelliJ IDEA built-in static analyzer (which is not available standalone).

TODO: maybe add more text? What else could be included?

Chapter 8

Acknowledgements

Big thanks to Nikolai Kudasov for help and consultation related to EOLANG, my other team members, including: Eugene Zouev, Egor Klementev, Ilya Miluoshin for collaborative work on J2EO, Yegor Bugayenko for releasing phi-calculus and EOLANG compiler, the open-source community for contributing to EOLANG repository and Rabab Marouf for teaching the Academic Writing course.