Abstract

Depend on above part

Chapter 1

Introduction

Object-oriented programming languages have been adopted widely over the past two decades. As of March 2022, the top five positions of the TIOBE index are occupied by Python, C, Java, C++ and C#. Four of these languages (with the exception of C) are considered object-oriented and, as the index suggests, are widely adopted and used in large-scale commercial products.

1.1 Properties of Object-Oriented Programming Languages

According to [1], *object-oriented* programming languages are the languages where the main unit of abstraction is an *object*. Objects encapsulate *data*, which are the values of some type. Some languages, e.g. Java and C++, distinguish between *primitive types*, which represent low-level constructs like numbers or boolean values, and *object types*, which represent a composite type. Objects may also contain operations on the said data, known as *methods*. Methods can take parameters and may return a value.

Objects should also obey certain definitive properties. As [1] suggests, ” the extent to which a particular language satisfies these properties defines how much of an object-oriented language it is.” These properties are:

- Encapsulation - an object should expose a well-defined interface through which it should be consumed. The irrelevant details of how an object implements this interface should be *hidden* from the consumer.
- Inheritance - is a mechanism through which objects can share functionality and extend the behavior of other objects. Inheritance is a complex mechanism and its implementation differs from language to language. As per [1], ”Inheritance enables programmers to reuse the definitions of previously defined structures. This clearly reduces the amount of work required in producing”.
- Polymorphism - a possibility to define operations on objects in such a way, that they can accept and return values of multiple types.
- Dynamic (or late [2]) binding - the implementation of the method to be run on an object is chosen at runtime. This implies that the implementation that is used during the runtime of a program may be *different* from that of a type that is known statically (i.e. at compile time)

1.2 Criticism

Together with increasing adoption, OO programming techniques and languages have gained a substantial amount of valid criticism. Mansfield [3] mentions most of these complaints, ultimately claiming that ”...with OOP-inflected programming languages, computer software becomes more verbose, less readable, less descriptive, and harder to modify and maintain”. Many of these criticisms are being turned into recommendations, such as the famous ”Design patterns: elements of reusable object-oriented software” [4]. However, such recommendations are not the part of the language specification and thus can not be enforced by the language compiler. This leads to these recommendations often being misinterpreted or overused, especially by beginners.

1.3 Analysis Tools

To mitigate this complexity and enforce good practices, developers have created a lot of software tools. These tools can be divided into two categories: **dynamic** analyzers and **static** analyzers.

Dynamic analyzers (also known as *profilers*) inspect the state of the program as it is being executed. Dynamic analyzers collect important information about the program execution, such as CPU utilization and memory consumption and present it in the human-readable form. This information is crucial in applications where the performance plays an important role. Unfortunately, the tools require the program under analysis to be executed, which can be expensive or even impossible, e.g. when the program is to be run on dedicated hardware.

On the contrary, **static** analyzers inspect the source code of the program (or one of its intermediate representations) *without executing it* to locate common errors, anti-patterns and deviations from the accepted style conventions. Executing such tools isn’t usually time-consuming or otherwise expensive, which is why they are a crucial part of continuous integration (CI) pipelines and integrated development environments (IDEs). Despite being prone to false positives, static analysis tools can pinpoint the location of the error with greater precision.

Unlike dynamic analyzers, static analyzers operate on the source code, which allows them to inspect the program from a higher-level perspective. This means that static analyzers can improve the error reporting of programming language compilers, discover more problems, and even automatically fix them.

Prior to analysis, many static analysis tools convert the source of the target language into some intermediate representation. This is done for several reasons. In general, this is done to extract the information from the source code that is relevant for analysis needs. Another common use case for employing an intermediate representation would be to make the static analyzer work with more than one target language. In this case the representation serves as a common ground for the various analyzers. The examples of intermediate representation are LLVM [5] and Jimple [6] (used in SOOT [7]).

1.4 Research Objective

In this thesis we present an implementation of a module for a static analyzer of object-oriented programs which takes the program representation in Elegant Objects (EO) [8] as an input and produces simple error messages as output. EO is an intermediate representation based on ϕ-calculus, a formal model that is intended to unify the varying semantics of object-oriented languages. It also claims to be a language with minimum verbosity, providing the minimum necessary set of operations. The combination of a strict formal ground and a reduced feature set make EO a powerful intermediate representation for a static analyzer that should be able to capture many bugs specific to OO programs. This thesis describes the proof-of-concept implementation of detecting the two defects of the ”fragile base class” [9] family: ”unanticipated mutual recursion” and ”unjustified assumption in modifier”.

The rest of this thesis is structured as follows: chapter 2 covers the existing work of finding bugs in OO programs, chapter 3 describes the semantics of EO and how it can represent object-oriented programs, chapter 4 describes the implementation of the analyzer, chapter 5 covers the evaluation of the implementation, including testing and benchmarks and, finally, chapter 6 concludes the thesis.

Chapter 2

Literature Review

This chapter presents oveview of the theoretical concepts that the implementation relies on. Section 2.3 briefly describes the relevant parts of φ-calculus: its syntax and semantics. Section 2.4 explains how φ-calculus maps to EO, the intermediate representation the analyzers operate on. Section 2.5 shows how to encode basic object-oriented constructs (classes, methods, inheritance) by means of EO.

2.1 Navigation

2.2 Methods & Criteria

2.3 φ-calculus

EO is a programming language that implements φ-calculus, a formal model for object-oriented programming languages initially introduced by Bugayenko [8]. In this thesis we use a refinement of φ-calculus proposed by Kudasov and Sim [10].

2.3.1 Objects and attributes

At the heart of φ-calculus lies the concept of **object**.

Definition 1 (Objects and attributes)**.** An **object** is a set of pairs $[\![n_0 \mapsto o_0, n_1 \mapsto o_1, \dots, n_i \mapsto o_i, \dots]\!]$, where n_i is a unique identifier and o_i is an object. Such pairs are known as **attributes**. The first element is the **attribute name** the second element is the **attribute value**. An empty set $[\![]\!]$ is also a valid object. An attribute where the second element is $[\![]\!]$ is called **void** or **free**. Otherwise, it is known as **attached**.

Attributes of object can be accessed by their names via the dot notation:

$$[\![x \mapsto y]\!].x \rightsquigarrow y$$

In this case, this would reduce to just object y, which is defined elsewhere. $\rightsquigarrow$ means ”is reduced to” or ”evaluates to”.

2.3.2 Application

Application can be used to create a new object where the values of the some or all free attributes are set. In other words, application can be used to create *closed* objects from *abstract* objects.

Definition 2 (Abstract and closed objects)**.** If an object has one or more free attributes it is called **abstract** or **open**. Otherwise, it is called **closed**.

For example, object a in 2.2 corresponds to a point in a two-dimensional space with coordinates $x = 1, y = 2$. The objects 1 and 2 can be defined in terms of φ-calculus, however the definition itself is out of the scope of this thesis.

$$point := [\![x \mapsto [\![]\!], y \mapsto [\![]\!]]\!] \tag{2.1}$$

$$a := point(x \mapsto 1, y \mapsto 2) \tag{2.2}$$

$$a \rightsquigarrow [\![x \mapsto 1, y \mapsto 2]\!] \tag{2.3}$$

2.3.3 Locators

The revisision of φ-calculus by Kudasov and Sim [10] also defines special objects called **locators**, which are denoted as ρ^i, where $i \in \mathbb{N}$. Locators allow objects to reference other objects relatively to the object where the locator is used. For example, this can be used to (but is not limited to) encode definition of attributes in terms of other attributes of this object. Suppose there is an object x:

$$x := [\![a \mapsto \rho^0.b, b \mapsto c]\!]$$

The expression $x.a$ would be reduced to the value of object c. This happens because $x.a$ references $x.b$ via ρ^0, which means the immediate enclosing object. In more complicated examples, like 2.4.

$$x := [\![a \mapsto [\![c \mapsto \rho^1.b]\!], b \mapsto d]\!] \tag{2.4}$$

$$x.a.c \rightsquigarrow d \tag{2.5}$$

ρ can be used to define attributes of inner objects in terms of attributes of outer objects, or even outer objects themselves.

2.3.4 φ-attribute

Objects can define a special attribute with name φ. This attribute redirects attribute access to its value when the enclosing object does not have an attribute with such a name (fig. 2.6).

$$a := [\![d \mapsto y]\!] \tag{2.6}$$

$$x := [\![\varphi \mapsto a, c \mapsto g]\!] \tag{2.7}$$

$$x.d \rightsquigarrow x.\varphi.d \rightsquigarrow y \tag{2.8}$$

If the attribute is present both in the object and its φ-attribute, the attribute in the object takes precedence:

$$\begin{aligned} a &:= [\![d \mapsto y]\!] \\ x &:= [\![\varphi \mapsto a, \mathbf{d} \mapsto g]\!] \\ x.d &\rightsquigarrow g \end{aligned}$$

In 2.7, Bugayenko [8] refers to object a as **decorated object**, where the ”decorated” part refers to the decorator pattern described in [4, Chapter 4]. This technique of extending an object is also known as *delegation* [11] in object-oriented languages.

2.3.5 A complex example

Tying everything together, figure 2.1 shows how φ-calculus can be used to compute Fibonacci numbers.

```
$fib := [\![$
    $n \mapsto [\![]\!],$
    $\varphi \mapsto \rho^0.n.less(n \mapsto 2).if($
     $ifTrue \mapsto n,$
     $ifFalse \mapsto$
     $fib(n \mapsto \rho^0.n.sub(n \mapsto 1))$
     $.add(n \mapsto fib(n \mapsto \rho^0.n.sub(n \mapsto 2)$
    $)$
    $)$
    $)$
    $]\!]$
```

Figure 2.1: Fibonacci numbers in φ-calculus

2.4 EO

EOLANG, or simply EO, is a programming language created by Bugayenko [8] which is a direct implementation of φ-calculus with some extensions. However, their implementation contains features that are irrelevant to the scope of this thesis. Moreover, there is a notable difference between Bugayenko version of EO and ϕ-calculus by [10] in the definition of locators (or ”parent objects”). In the work by Bugayenko locators are *attributes*, whereas in [10] they are *objects*. In this thesis, similarly to the φ-calculus, we are going to use a different version of EO which is a direct translation of the calculus defined in 2.3. The table of translation is shown in figure 2.2.

	φ-calculus	EO

Objects	$obj := \llbracket a \mapsto x, b \mapsto y \rrbracket$	`[] > obj`
		`  x > a`
		`  y > b`

Free Attrubutes	$point := \llbracket x \mapsto \llbracket\rrbracket, y \mapsto \llbracket\rrbracket \rrbracket$	`[x y] > point`
Application	$a := point(x \mapsto 1, y \mapsto 2)$	`point 1 2 > a`

φ-attribute	$x := \llbracket \varphi \mapsto a, c \mapsto g \rrbracket$	`[] > x`
		`  a > @`
		`  g > c`

Fibonacci example	Fig. 2.1	`[n] > fib`
		`  ($.n.less 2).if > @`
		`    n`
		`    (fib ($.n.sub 1)).add`
		`      (fib ($.n.sub 2))`

ρ^0	ρ^0	$
ρ^1	ρ^1	^
ρ^3	ρ^3	^.^.^

Figure 2.2: Mapping φ-calculus to EO

2.5 Describing object-oriented programs with EO

Before analyzing programs written in object-oriented programming languages, it is necessary to translate them into EO while preserving the semantics of the original language. This chapter presents a simplified version of such an encoding that is assumed by analyzers described in this thesis.

2.5.1 Classes

Classes are modelled as closed EO objects. Class-level (i.e. ”static”) attributes become attributes of the class object. Constructor is represented by an attribute-object ”new” of the class object. This object may take parameters to produce an instance of the object.

All instance attributes and methods are defined inside the object returned by the ”new” object. Inheritance is modelled as decoration in EO. So, a full example of EO translation would look like this. Class instances (a.k.a objects in Java) are created by applying the ”new” object to the required parameters.

2.5.2 Methods

Methods are modelled as EO objects, similarly to classes. These objects can take parameters. Instance methods are required to accept a special **self** attribute in addition to other parameters. This parameter is used to pass an instance of the object calling the method (hence the name - ”self”). ”self” parameter can be used to call instance methods inside other instance methods.

The return value of the method is represented by the value of the φ attribute (”@” symbol in EO). In order to call the instance method we need to instantiate the object first. Then we can call the method by accessing the instance's attribute with the method name and passing the instance object to it as the first argument.

2.6 Fragile Base Class Problem

2.6.1 Unanticipated Mutual Recursion

Unanticipated mutual recursion is a problem that occurs as a result of unconstrained inheritance. Suppose we have an object named Base with two methods - *f* and *g*. Method *g* calls method *f*, whereas *f* does not.

Then, there is a class called Derived that extends Base and redefines the method *f* in a way that it calls *g*. When we call a method *f* on an instance of Derived, we get a stack overflow error: method *f* calls method *g*, method *g* calls method *f* and so on (figure 2.3).

It is important to note that we are not interested in detecting mutual recursion between the two methods of the same class. We are only interested the cases where mutual recursion occurs as a result of redefining one of the methods of the superclass. The example in fig. 2.4 shows the class with two mutually-recursive methods ”isOdd” and ”isEven”. In this case the recursion is anticipated and necessary, so it is not a defect.

2.6.2 Unjustified Assumption in Subclass

This defect [9, Section 3.3] occurs when the superclass is refactored by *inlining* the calls to the method that can be redefined by the subclass. The term inlining refers to replacing the method call with its body. Consider an example (Fig. 2.5). Class *M* extends class *C*, redefining method *l* to weaken its precondition. Con-

```java
class Base {

    int f(int v) {
        return v;
    }

    int g(int v) {
        return this.f(v);
    }
}

class Derived extends Base {
    @Override
    int f(int v) {
        return this.g(v);
    }
}
```

(a) Java

```
[] > base
  [self v] > f
    v > @
  [self v] > g
    self.f > @
      self
      v
[] > derived
  base > @
  [self v] > f
    self.g > @
      self
      v
```

(b) EO

Figure 2.3: Example of unanticipated mutual recursion

```
class NumericOps {
    boolean isEven(int n) {
        if (n == 0) {
            return true;
        } else {
            return
                this.isOdd(n - 1);
        }
    }

    boolean isOdd(int n) {
        if (n == 0) {
            return false;
        } else {
            return
                this.isEven(n - 1);
        }
    }

}
```

(a) Java

```
[] > numeric_ops
  [self n] > is_even
    ($.n.eq 0).if > @
      1
      $.self.is_odd
        $.self
        ($.n.sub 1)
  [self n] > is_odd
    ($.n.eq 0).if > @
      0
      $.self.is_even
        $.self
        ($.n.sub 1)
```

(b) EO

Figure 2.4: Example without unanticipated mutual recursion.

sequently, the precondition in method *m* of class *M* is also weakened, because it relies on calling the method *l*.

Now, suppose that class *C* comes from some external library, and class *M* is defined in the user code. Library maintainer decides to refactor class *C* by inlining the call to *l* in method *m* (Fig. 2.6). Observe what happens to the class *M*. Now that *m* in class base has an assert, the redefinition of method *n* in class *M* has its precondition strengthened as compared to its version in class *C*. Therefore, the seemingly safe refactoring in base class broke the invariants in the subclasses. The name of the defect come from the fact that the subclasses usually *M assume* that the method *m* should be implemented in terms of method *l*. The examples in fig. 2.5 and 2.6 show that such an assumption is indeed not justified, and the maintainers of class *C* can change it as they deem fit.

```java
class C {
    int l(int v) {
        assert (v < 5);
        return v;
    }

    int m(int v) {
        return this.l(v);
    }

    int n(int v) {
        return v;
    }
}

class M extends C {
    int l(int v) {
        return v;
    }

    int n(int v) {
        return this.m(v);
    }
}
```

(a) Java

```
[] > c
  [self v] > l
    seq > @
      assert (v.less 5)
      v
  [self v] > m
    self.l self v > @
  [self v] > n
    v > @

[] > m
  c > @
  [self v] > l
    v > @
  [self v] > n
    self.m self v > @
```

(b) EO

Figure 2.5: Example of unjustified assumption in subclass (before revision)

```
class C {
    int l(int v) {
        assert (v < 5);
        return v;
    }

    int m(int v) {
        assert (v < 5);
        return v;
    }

    int n(int v) {
        return v;
    }
}

class M extends C {
    int l(int v) {
        return v;
    }

    int n(int v) {
        return this.m(v);
    }
}
```

(a) Java

```
[] > c
  [self v] > l
    seq > @
      assert (v.less 5)
      v
  [self v] > m
    self.l self v > @
  [self v] > n
    v > @

[] > m
  c > @
  [self v] > l
    v > @
  [self v] > n
    self.m self v > @
```

(b) EO

Figure 2.6: Example of unjustified assumption in subclass (after revision)

Chapter 3

Methodology

Chapter 4

Implementation

This chapter analyzes specifics of Odin (short for Object Dependency INspector) - a static analyzer of EO source code. Section 4.1 covers the tools and technologies used in the project. Section 4.2 gives a brief overview of the project file structure. Section 4.3 goes over the implementation of EO parser used in the project. Section 4.4 discusses the ways we can represent the elements of object-oriented programs in EO. Sections 4.5 and 4.6 describe the implementations of the analysis algorithms applied to structured EO code. Finally, section 4.7 summarizes this chapter.

4.1 Development Environment

Odin is a project written entirely in **Scala** [1] - a modern programming language with support for high-level concepts such as structural pattern matching [12] and algebraic data types [13]. Scala is compiled into Java Virtual Machine (JVM) byte code. This allows Odin to be used as a library in any other project compatible with JVM, be it Scala or Java. In addition, programs compiled to JVM byte code

[1] https://www.scala-lang.org/

can be run without changes on any device that can run Java Virtual Machine.

The project uses a build tool called **sbt** [2], which allows compiling multiple Scala modules at once. A distinctive feature of **sbt** is the ability to cross-compile Scala code so that it is compatible with many versions of Scala and Java. It also supports a variety of plugins that improve the development process. The plugins used by Odin are **scalafmt** [3], an automatic source code formatter, and **scalafix** [4] , a linter and code analyzer with support for project-wide refactorings.

Odin is published as a **JAR** [5] and can be downloaded from the **Maven Central** repository [6].

The source code of the project is available on **Github** [7]. It also provides the instructions on how to launch and contribute to the project.

4.2 Module Structure

Odin is a project consisting of multiple modules. The main modules are:

- Core, which contains the definition for EO AST (Abstract Syntax Tree). This AST is used as an input to all analysis algorithms.
- Analyses, which contains the implementations of various analyzers.
- Backends, which contains algorithms that transform EO AST into something else. The only backend so far is a plain text backend: it transforms EO AST into its syntactically correct equivalent in EO source code. This backend can

[2] https://www.scala-sbt.org/

[3] https://scalameta.org/scalafmt/

[4] https://scalacenter.github.io/scalafix/

[5] https://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html

[6] https://search.maven.org/search?q=g:org.polystat.odin

[7] https://github.com/polystat/odin

also be interpreted as a pretty-printer of EO code and is widely used as such in other modules.

- Parser, which contains a parser (also known as a syntactic analyzer) of EO source code. It is used to convert different EO representations (e.g. plain text or XML encoding) into the EO AST defined in Core module.

4.3 Data Structures

4.3.1 EO Syntax Tree

This is a data structure that is used to model the syntactic structure of EO which is used as a starting point for the extraction of more high-level concepts, such as class-objects, method-objects and method calls. It describes EO following the syntax specification from the paper by Bugayenko [8], with slight deviations to account for the specifics of the underlying refined φ-calculus described in [10].

In order to create the parser from the EO code to EO syntax tree we used **cats-parse** [8], a monadic parser combinator [14] library for Scala.

EO syntax tree is an immutable polymorphic data structure defined *à la carte* [15] (Fig. 4.1). Since the tree is immutable, it can only be altered by constructing the new version, where the old parts of the tree are replaced with new ones. The transformations that construct these new versions of the data structure are known as *optics* [16]. We used the *monocle* [9] library for Scala to simplify the generation of the optics that modify the EO syntax tree.

[8] https://github.com/typelevel/cats-parse

[9] https://www.optics.dev/Monocle/

```
final case class Name(name: String)

// Expression
sealed trait EOExpr[+A]

// Object
sealed case class EOObj[+A](
  freeAttrs: Vector[Name],
  varargAttr: Option[Name],
  bndAttrs: Vector[EOBndExpr[A]],
) extends EOExpr[A]

// Application
sealed trait EOApp[+A] extends EOExpr[A]

sealed case class EOSimpleApp[+A](
    name: String
) extends EOApp[A]

sealed case class EOSimpleAppWithLocator[+A](
  name: String,
  locator: Int
) extends EOApp[A]

sealed case class EODot[+A](
  src: A,
  name: String
) extends EOApp[A]

sealed case class EOCopy[+A](
  trg: A,
  args: NonEmptyVector[EOBnd[A]]
) extends EOApp[A]
```

Figure 4.1: EO syntax tree definitions (abridged).

```
final case class ObjectTree[A](
  info: A,
  nestedObjs: Map[Name, ObjectTree[A]]
)
```

Figure 4.2: Object tree

4.3.2 Object Tree

Object Tree is a data strcuture that captures the relationships between objects in an EO program. It is a refinenement of the EO syntax tree, which contains the elements of an EO program relevant to subsequent analysis steps: class-objects, method-objects, extension clauses and method calls.

EO object tree is also a recursively-defined polymorphic data structure (Fig. 4.2). The type parameter *A* represents the information that is stored for each object in the tree. This information is stored in *info* field of the tree. The field called *nestedObjs* is stores the information about all the nested class-objects. Nested objects are the class-objects that are defined as the attributes of other class objects, just like nested classes in Java. The information about one of the nested objects can be accessed by the key which is of type *Name*. This name identifies the object uniquely because the object can not contain two attributes with the same name.

4.3.3 ObjectInfo

The first and the most generally-applicable type we use in place of type parameter *A* is *ObjectInfo* (Fig. 4.3). This type also has two type parameters. The first one, *P*, is responsible for holding the information about the decorated object (or simply parent). The first traversal can only gather the name of the decorated object.

The second type parameter, *M*, is the type used to store the information about each of the methods. The information captured during the first traversal of EO syntax tree (Fig. 4.4) can be summarised as follows:

- *selfArgName* - the name of the free attribute of the method object that is used to capture the calling object.
- *body* - the EO syntax tree node that hold the body of the method.
- *depth* - how deeply the method object is nested in an EO program. For toplevel objects this attribute is 0. For method objects (that are always defined in the class-objects), this attribute is equal to the depth of the class-object plus one.
- *calls* - a sequence of method calls in the method definition. The *Call* type stores all the necessary information to identify and traverse all the call within the method body:
    - depth - depth of the object where the call is located. This value is equal to the depth of the method-object + the relative depth of the method-local object containing the call.
    - *methodName* - a simple name of the method-object where the call is located.
    - *callSite* - an optic [16] which extracts the location of the method-local object where the call is located.
    - *callLocation* - an optic [16] that extracts the location of the EO syntax tree node that defines the method call.
    - *args* - the EO syntax tree nodes, which correspond to the arguments of the call, including the **self** argument.

```
final case class ObjectInfo[P, M](
  name: Name,
  fqn: FQName,
  depth: Int,
  parentInfo: Option[P],
  methods: Map[Name, M],
)
```

Figure 4.3: ObjectInfo

```
final case class MethodInfo(
    selfArgName: String,
    calls: Vector[Call],
    body: EOObj[EOExprOnly],
    depth: Int,
)

final case class Call(
  depth: BigInt,
  methodName: String,
  callSite: PathToCallSite,
  callLocation: PathToCall,
  args: NonEmptyVector[EOBnd[EOExprOnly]]
)
```

Figure 4.4: MethodInfo and Call

4.4 Partial object tree

The *ObjectInfo*, where *P* is the name of the parent object and *M* is *MethodInfo* can be called the *partial object tree*:

```
type PartialObjectTree = ObjectTree[

  ObjectInfo[ParentName, MethodInfo]

]
```

4.5 Complete object tree

The so-called *complete object tree* is defined as the following type alias:

```
type CompleteObjectTree =
  ObjectTree[
    ObjectInfo[
      LinkToParent,
      MethodInfo
    ]
  ]
```

The only thing that distinguishes it from the *partial object tree* is that the parent name is replaced with the special *LinkToParent* type. This type is also an optic [16] and is essentially a function of the type:

```
  val linkToParent:
    CompleteObjectTree => CompleteObjectTree
```

It takes the whole object tree of the program and returns the object which represents the decorated object of the object where the link is located.

4.6 Detecting Unanticipated Mutual Recursion

4.6.1 Proposed solution

The solution to the problem lies in detecting the cycles in the call-graphs of all the objects. For each class-object in the program, do the following:

1. Detect the decorated class-object, all methods, and for each method in the class detect all the methods it calls. If the method that is called exists in the

class-object, mark it as *resolved*. Otherwise, mark it as *partially-resolved*. The set of mappings between the methods of the class and the methods that each of the methods calls is considered a *partial call-graph* of the object.

2. After that the tree is traversed again to convert all the partially-resolved calls to fully resolved calls. To do that we need to calculate the *complete call-graph* of the object, which contains the methods from the object itself, as well as the methods from the decorated object. This is done by *extending* the partial call-graph of the decorated object with the partial call-graph of the decorating objects. Hereinafter we use the terms **child** and **parent** to refer to the decorating object and the decorated object respectively. The extension procedure is defined as follows:

    (a) if the method is present in the parent call-graph, but is absent in the child call-graph, it is left as is.

    (b) if the method is present in the child call-graph but does not exist in the parent call-graph, it is added to the parent call-graph.

    (c) if the method is present both in the child call-graph and the parent call-graph, all the occurrences of the method in the parent call-graph are replaced by the child's version of the method.

3. After the object's call-graph is resolved, perform the depth-first search [17] to find the cycles in the complete call-graph. After all the cycles are found, exclude the cycles that contain only the methods from the same object.

```
type ResolvedCallGraph =
  Map[Methodname, Set[MethodName]]
type ObjectTreeWithResolvedCallgraphs =
  ObjectTree[ResolvedCallGraph]
```

Figure 4.5: *ObjectTreeWithResolvedCallgraphs*.

4.6.2 Implementation

The first step of the algorithm in 4.6.1 is covered by converting the EO code into a *complete object tree*. Then the object tree is converted into a variation of the object tree where all the calls are resolved. This variation is called *ObjectTreeWithResolvedCallgraphs* (Fig. 4.5). For each object, only its fully resolved call-graph is stored. The final step traverses the tree and collects all the cycles in a list of special objects of type *CallChain*. This object represents the sequence of fully-qualified method names that when called would never terminate. Finally, each of such call chains are presented to the user in the form of human-readable strings.

4.7 Detecting Unjustified Assumption in Subclass

4.7.1 Proposed Solution

We propose the following approach for detecting the methods where inlining of the calls may lead to breaking changes in subclasses:

1. An *initial* representation of the program is produced. This representation is a tree-like data structure which preserves the nesting relations between objects. So, the objects which contain other objects are the roots of their respective subtrees, whereas the container objects are the subtrees or leaves.

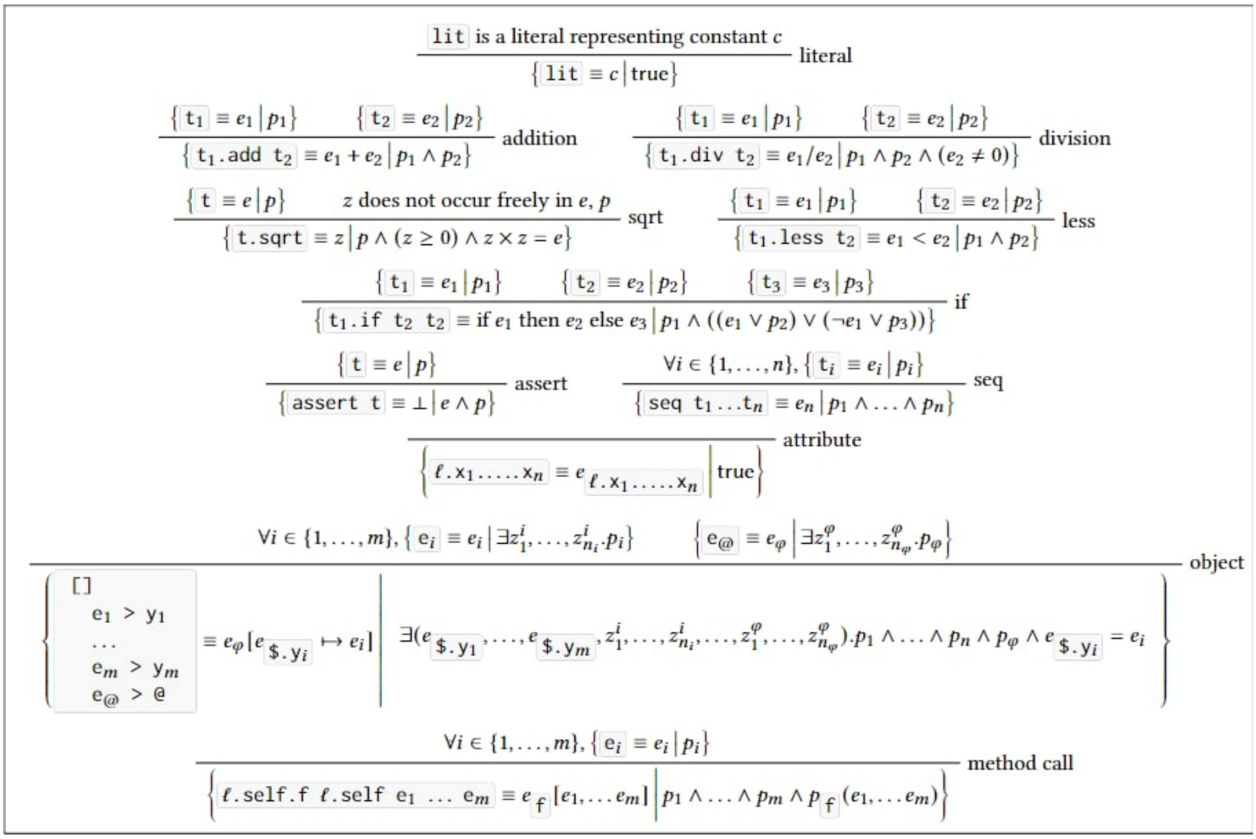
2. We produce a *revision* of the initial program representation where all the calls to the methods are inlined.

3. In both versions, for each of the class-objects, for each method in the class-object, a set of*properties* is inferred. These properties can be thought of as an implicit contract [18] of each method. In addition to the implicit properties, the explicit properties which come in the form of *assert* statements in the source code are also taken into account. In order to infer the properties of the method, partial interpretation of its body is performed. The interpretation is limited to basic numeric operations, numeric and boolean values and method calls. The inference rules are described in greater detail in fig. 4.6.

4. After all the properties are inferred, the following predicate should hold true for both the initial and the revised versions:

   $$P_{init} \Rightarrow P_{rev}$$

   If it doesn't hold for some class-object, it means that the revision of one of its superclasses introduces a breaking change, which weakens the precondition of some its methods.



$$\frac{\texttt{lit} \text{ is a literal representing constant } c}{\{\texttt{lit} \equiv c \mid \text{true}\}}\ \text{literal}$$

$$\frac{\{\texttt{t}_1 \equiv e_1 \mid p_1\} \qquad \{\texttt{t}_2 \equiv e_2 \mid p_2\}}{\{\texttt{t}_1\texttt{.add t}_2 \equiv e_1 + e_2 \mid p_1 \wedge p_2\}}\ \text{addition} \qquad \frac{\{\texttt{t}_1 \equiv e_1 \mid p_1\} \qquad \{\texttt{t}_2 \equiv e_2 \mid p_2\}}{\{\texttt{t}_1\texttt{.div t}_2 \equiv e_1/e_2 \mid p_1 \wedge p_2 \wedge (e_2 \neq 0)\}}\ \text{division}$$

$$\frac{\{\texttt{t} \equiv e \mid p\} \qquad z \text{ does not occur freely in } e, p}{\{\texttt{t.sqrt} \equiv z \mid p \wedge (z \geq 0) \wedge z \times z = e\}}\ \text{sqrt} \qquad \frac{\{\texttt{t}_1 \equiv e_1 \mid p_1\} \qquad \{\texttt{t}_2 \equiv e_2 \mid p_2\}}{\{\texttt{t}_1\texttt{.less t}_2 \equiv e_1 < e_2 \mid p_1 \wedge p_2\}}\ \text{less}$$

$$\frac{\{\texttt{t}_1 \equiv e_1 \mid p_1\} \qquad \{\texttt{t}_2 \equiv e_2 \mid p_2\} \qquad \{\texttt{t}_3 \equiv e_3 \mid p_3\}}{\{\texttt{t}_1\texttt{.if t}_2 \texttt{ t}_2 \equiv \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid p_1 \wedge ((e_1 \vee p_2) \vee (\neg e_1 \vee p_3))\}}\ \text{if}$$

$$\frac{\{\texttt{t} \equiv e \mid p\}}{\{\texttt{assert t} \equiv \bot \mid e \wedge p\}}\ \text{assert} \qquad \frac{\forall i \in \{1, \ldots, n\}, \{\texttt{t}_i \equiv e_i \mid p_i\}}{\{\texttt{seq t}_1 \ldots \texttt{t}_n \equiv e_n \mid p_1 \wedge \ldots \wedge p_n\}}\ \text{seq}$$

$$\frac{}{\left\{\ell\texttt{.x}_1\texttt{.}\ldots\texttt{.x}_n \equiv e_{\ell\texttt{.x}_1\texttt{.}\ldots\texttt{.x}_n} \,\middle|\, \text{true}\right\}}\ \text{attribute}$$

$$\frac{\forall i \in \{1, \ldots, m\}, \{\texttt{e}_i \equiv e_i \mid \exists z_1^i, \ldots, z_{n_i}^i . p_i\} \qquad \left\{\texttt{e}_@ \equiv e_\varphi \,\middle|\, \exists z_1^\varphi, \ldots, z_{n_\varphi}^\varphi . p_\varphi\right\}}{\left\{\begin{array}{l}\texttt{[]} \\ \quad \texttt{e}_1 > \texttt{y}_1 \\ \quad \ldots \\ \quad \texttt{e}_m > \texttt{y}_m \\ \quad \texttt{e}_@ > @\end{array} \equiv e_\varphi[e_{\texttt{\$.y}_i} \mapsto e_i] \,\middle|\, \exists(e_{\texttt{\$.y}_1}, \ldots, e_{\texttt{\$.y}_m}, z_1^i, \ldots, z_{n_i}^i, \ldots, z_1^\varphi, \ldots, z_{n_\varphi}^\varphi).p_1 \wedge \ldots \wedge p_n \wedge p_\varphi \wedge e_{\texttt{\$.y}_i} = e_i\right\}}\ \text{object}$$

$$\frac{\forall i \in \{1, \ldots, m\}, \{\texttt{e}_i \equiv e_i \mid p_i\}}{\left\{\ell\texttt{.self.f } \ell\texttt{.self e}_1 \ \ldots \ \texttt{e}_m \equiv e_{\texttt{f}}[e_1, \ldots e_m] \,\middle|\, p_1 \wedge \ldots \wedge p_m \wedge p_{\texttt{f}}(e_1, \ldots e_m)\right\}}\ \text{method call}$$

Figure 4.6: Rules for property inference in detection of unjustified assumption in subclass.

Chapter 5

Evaluation and Discussion

This chapter provides the evaluation of the resulting implementation. Section 5.1 outlines the limitations of the EO-based static analysis. Section 5.2 describes how the analyzers were tested. Finally, section 5.3 describes the result of comparing EO-based static analyzers with their counerparts for other programming languages.

5.1 Limitations

5.1.1 General

The analyzer provided minimal information about the location of discovered errors. This is especially important if the analyzer is to be used in the integrated development environments.

The analyzer only works on single files. If EO objects are spread across multiple files, the current implementation would be able to analyze only one file at a time. There is a minimal support for objects imported from other EO files, i.e. analyzer acknowledges their existence and does not report them as missing.

```
[] > a
  b > @

[] > b
  a > @
```

Figure 5.1: A simple example of the cycle in the inheritance chain. Object *a* extends *b*, which in turn extends object *a*.

However, bodies of object imported from other files can not be accessed, therefore no meaningful analysis can be performed if one of the used objects comes from another file.

5.1.2 Unanticipated Mutual Recursion

The implementation of does not do any path-dependent analysis, therefore it may produce false-positives in case when the call to the mutually-recursive method is unreachable, e.g. when guarded by a statement which can only be false.

The current implementation of the mutual recursion analyzer also contains a bug which causes it to crash with a stack overflow exception when the program contains a cycle in the inheritance chains. The presence of the inheritance chain means that there is such a class-object that extends an object which directly or indirectly extends the first object.

5.1.3 Unjustified Assumption in Subclass

The analyzer relies on the external SMT-solver called Princess [19]. The solver has some limitations when it comes to the support of SMTLIB-2 [20] format. A lot of effort was spent working around the peculiarities of the solver interface.

This limits the portability of the implementation if a different SMTLIB backend is to be chosen.

The current implementation of the analyzer supports only a limited set of types. This limitation is directly imposed by the lack of a static type checker in EO. Specifically, all method parameters (excluding **self**) and their return methods are assumed to be of integer numeric type. This imposes significant limitations on the type of constraints that can be decided by the solver.

Finally, the complexity of the constraints the solver needs to decide grows linearly with the size of the program. While the complexity of the solver operation is largely unknown, it would be safe to estimate that the execution time of the analyzer would be rather slow on the large programs with a lot of constraints.

5.2 Testing

The implementation is largely covered by hand-written integration tests. For mutual-recursion, the property-based testing technique [21] was also applied to ensure the extensive coverage of the input domain.

5.2.1 Integration Testing

Integration testing or end-to-end testing [22, Chapter 7] describes an approach to testing when one testcase covers the functionality of the entire software system. In our case, the modules under tests where the analyzers, the input was the EO code with or without the respective defects and the expected output was the error messages produced by the respective analyzer, which is represented by a list of strings.

To execute the tests we used a testing framework for Scala called Scalatest [1]. To represent the individual test case we used a the following case class definition:

```
case class TestCase(
    label: String,
    code: String,
    expected: List[String]
  )
```

The *label* is a short description of the test case used mostly for human readability. *code* field holds the code to be analyzed, while the *expected* field holds the errors that are supposed to be detected by the analyzer in the *code* field.

All the test cases for a particular analyzer are divided into two groups: one for test cases where the input code contains errors, the other for test cases where the input code does not contain errors. These groups are each represented by a testsuite-local variable of type **List[TestCase]**:

```
    val testCasesWithErrors: List[TestCase] =
        List(casewithErrors1, ..., caseWithErrorsN)
    val testCasesWithoutErrors: List[TestCase] =
        List(caseWithoutErrors1, ..., caseWithoutErrorsN)
```

Each test group is then run using a driver function called *runTests*. This function registers the test in the Scalatest test suite, executes the analyzer on the input code to obtain the errors and then compares the obtained errors with the expected using the assert statement:

```
def runTests(tests: List[TestCase]): Unit =
```

[1] https://www.scalatest.org/

```
tests.foreach {
  case TestCase(label, code, expected) =>
    registerTest(label) {
      val obtained = analyze(code).unsafeRunSync()
      assert(obtained == expected)
    }
}
```

5.2.2 Property-based Testing

Property-based testing is a variation of random testing that can be used when there exists a set of well-defined properties (or predicates) that should be satisfied for a set of possible inputs of the function being tested. This predicate is then being evaluated on the randomly generated input data. The key difference between property-based testing and the conventional testing approach, like the one described in 5.2.1, is the fact that the random input data is generated *automatically*, which covers a large effective subset of the input domain that would otherwise be impractical or impossible to perform manually.

We decided to apply this approach to testing the mutual recursion analyzer as follows:

1. 1. Create a generator that randomly generates the instances of *ObjectTreeWithResolvedCallgraphs* (Fig. 4.5). The generation process keeps only those trees that contain the multiobject recursion cycles. These cycles are stored as *CallChain* objects and captured as expected output.

2. These trees are converted into string with the respective EO code by a simple pretty-printer.

3. The analyzer is then run on the generated strings to obtain the *CallChain* objects. The obtained objects are then compared with the ones before pretty-printing.

So, the property that we are trying to test can be summarised as follows:

Definition 3. The cycles found in the randomly-generated *ObjectTreeWithResolvedCallgraphs* should be the same as the ones found by the analyzer which was given the pretty-printed version of said *ObjectTreeWithResolvedCallgraphs*.

The current version of the CI pipeline for the repository runs 1000 such randomly generated tests per run.

5.3 Comparisons

Chapter 6

Conclusion

…