Detecting unjustified assumptions in subclasses via EO representation

Anonymous Author(s)

ABSTRACT

Elegant Objects (EO) is a programming language based on ideas of pure objects and the Decorator pattern. It has been suggested by Bugayenko as an intermediate representation for object-oriented programs. This paper presents a version of dynamic dispatch modelled in EO and formulates a problem of unjustified assumptions in decorator objects, which parallels similar problem in subclasses. Then, we introduce an approach to detect such problems in EO programs via method inlining and limited property inference. Finally, we discuss prototype implementation of this approach in Scala programming language.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**; **Object oriented languages**.

KEYWORDS

object-oriented programming, elegant objects, static analysis, anti-patterns

ACM Reference Format:

Anonymous Author(s). 2018. Detecting unjustified assumptions in subclasses via EO representation. In *Proceedings of The 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX


```
class A {
  float f(float x){ return Math.sqrt(x); }  // x ≥ 0
  float g(float y){ return this.f(y - 1); } // y ≥ 1
  float g_inlined(float y){ return Math.sqrt(y-1); }
  float h(float z){ return z; } // no restrictions
}

class B extends A {
  float f(float x){ return x*x; }
  float h(float z){ return this.g(z); }
}
```

Figure 1: Unjustified assumptions in a subclass in Java.

1 INTRODUCTION

Modern object-oriented languages are expressive tools for software engineers, but many of them do not fully isolate the implementation of a class. *Fragile base class problem* [6] is a a category of problems, where modification of a class is unsafe unless subclasses are also updated accordingly. Open recursion and method overriding lead to many of fragile base class problems.

A subcategory of fragile base class is concerned with unjustified assumptions in subclasses regarding method dependencies in superclasses. An example of such a problem is presented in Figure 1. In this example, method `B.h` has no restrictions of input parameter `z`. However, after refactoring the base class `A` by means of inlining the definition of method $A.f$ in the body of method $A.g$, the semantics of method $B.h$ changes. In particular, after inlining, $B.h$ imposes a restriction $z \geq 1$ on its formal argument z. Observe that, assuming objects `a` of type `A` and `b` of type `B`, we now have the following:

(1) **`class B`** is a subclass of **`class A`**;
(2) `a.h(z)` works for any **`float`** value of `z`;
(3) `b.h(z)` works only for **`float`** values of `z` such that $z \geq 1$.

This is a problem for two reasons. First, such arrangement violates Liskov substitution principle — it is no longer safe to pass an object of **`class B`** to an algorithm that expects an object of **`class A`**. Second, inlining $A.f$ in the body of $A.g$ did not affect the behaviour of instance of **`class A`**, but changed that of `B`, due to an unjustified assumption about self-calls in implementation of **`class A`**. In this paper, we are concerned with the second problem, which is described by Mikhajlov and Sekerinski [6].

Library designers are especially concerned with such problems as they do not control subclass definitions in the user code. Bloch [2], Szyperski et al. [8], and others advocate for delegation by wrapping base class instance in its original state and explicitly forwarding control when necessary. Bloch [2] also recommends that library designers make their classes **`final`** to disable the possibility of inheritance altogether.

Bugayenko [3] introduced the EO programming language, capitalizing on the idea of using composition over inheritance as a primary structuring tool. Having one simple feature like decoration, makes analysing the code structure easier. In particular, reformulation of some fragile base class problems in terms of EO becomes fairly straightforward as we present in Section 3.

In this paper, we consider EO as an intermediate representation of an object-oriented program. We reformulate the unjustified assumptions problem for EO, and present an approach for detecting such problems.

Our specific contributions and paper structure are the following:

(1) In Section 2, we demonstrate how dynamic dispatch can be expressed in EO, in particular introducing the concept of method in EO;

ESEC/FSE 2022, 14 - 18 November, 2022, Singapore

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

(2) In Section 3 we introduce a concept of method properties and reformulate the unjustified assumptions problem for EO;
(3) In Section 4 we present an algorithm for detecting unjustified assumptions, based on method inlining and method property inference;
(4) In Section 5 we discuss the implementation of the suggested approach in Scala programming language;
(5) In Section 6 we discuss achieved results, limitations and potential future work.

2 ELEGANT OBJECTS AND CLASSES

EO is an untyped[1] purely object-oriented programming language with decoration as a primary tool for object modification. It has been suggested by Bugayenko [3] as a candidate intermediate language to represent object-oriented programs. Bugayenko [4] has outlined general ideas for translating various object-oriented concepts to EO, and in this paper we assume those ideas apply unless otherwise stated.

One feature of EO is *locators*. This feature allows one to reference outer objects in a nameless way: `$` refers to the current object, `^` — to the parent object (one nesting level up), `^.^` — parent of a parent (two levels up), and so on. In the original syntax of EO locators are often optional, meaning that one can use an attribute name without a locator, implicitly implying the closest object with given attribute. Bugayenko [3, Section 3.8] mentions that locators can always be recovered. In this paper, we will assume that all programs have explicit locators.

Bugayenko [4, Section 1.4] sketches translation of classes to EO. The general idea is to translate classes to objects that can produce new objects (instances). In this paper, we assume that (non-static) methods of a class take an extra argument — the object itself, as its first argument. We reserve identifier `self` for this special argument.

Example 2.1. Consider the following Java program:

```java
class Book {
  String title;
  public Book(String title) { this.title = title; }
  public String path() {
    return "/books/" + this.title + ".txt";
  }
  public String rename(String new_title) {
    this.title = new_title;
    return this.path();
  }
}
Book book = new Book("War and Peace");
```

The following is the corresponding EO code. Note the use of `self` in places that correspond to method definitions as well as method calls of the Java program:

```
[] > Book
  [] > new
    memory > title
    [self] > path
      "/books/".append > @
        $.self.title
        ".txt"
    [self new_title] > rename
      seq > @
        $.self.title.write $.new_title
        $.self.path $.self
  [self title] > Book_constructor
    seq > @
      $.self.title.write $.title
      $.self

Book.Book_constructor > book
  Book.new
  "War and Peace"
```

In the next section, we focus specifically on methods and method calls. For the rest of the paper, we will not consider particular details of translating classes to EO.

2.1 Methods

We are mostly interested in EO analogue of virtual methods. The general practice is to model functions in EO is by using objects with void attributes as parameters and the special decorator attribute `@` used for the return value. Virtual methods are then modelled by introducing an extra parameter (void attribute) `self`. More precisely, the following definition formalizes the syntactic form of a virtual method:

Definition 2.2. An attached attribute `f` of an object term `t` is called a **method**, if (i) it is attached to an object term with void `self` and attached `@` attributes, and (ii) there are no references to its `@` attribute in any of its attached attributes. If `f` has n void attributes, not counting `self`, then we say that `f` has **arity** n. Attached attributes of `f`, excluding `@`, are called **local definitions** of `f`. We call object `t` the **owner object** of method `f`.

Example 2.3. In the following example attribute `obj.f` is a method with arity 1, local definition `y`, and method owner `obj`:

```
[] > obj
  [self x] > f
    $.x.add 1 > y
    ^.avg $.y $.x > @
  [a b] > avg
    ($.a.add $.b).div 2 > @
  [self] > g
    $.@ > original
    3 > @
```

Note that attribute `obj.avg` is not a method, since it does not have `self` void attribute, and `obj.g` is not a method since `@` is referenced in one of its local definitions (in `original`).

Definition 2.4. A term of the form

$$\ell\texttt{.self.g}\quad \ell\texttt{.self}\quad \texttt{t}_1\ \ldots\ \texttt{t}_n$$

is called a **method call** when ℓ is a locator (such as `$`, `^`, `^.^`). We call the object term referenced by ℓ the **owner object of method call** $\ell\texttt{.self.g}\ \ \ell\texttt{.self}\ \ \texttt{t}_1\ \ldots\ \texttt{t}_n$.

[1] Here, by "untyped" we mean that EO does not have a static type system.

Example 2.5. Consider the following EO program:

```
[] > a
  [self x y] > f
    $.x.add $.y > z
    $.z.mul $.z > @
  [self x] > g
    $.self.f $.self $.x > @
[] > b
  ^.a > @
  [self x y] > f
    $.self.g $.self ($.x.add $.y) > @
  [self z] > h
    ^.@.g ^.@ $.z > @
```

Here we have the following method calls:

(1) `$.self.f $.self $.x` is a method call with `a.g` as its owner object;
(2) `$.self.g $.self ($.x.add $.y)` is a method call with `b.f` as its owner object;
(3) `^.@.g ^.@ z` is not a method call, as it does not rely on `self` attribute.

3 UNJUSTIFIED ASSUMPTIONS IN DECORATED OBJECTS

Mikhajlov and Sekerinski [6] classify several fragile base class problems. One of this problems involves unjustified assumptions in subclasses regarding the way methods depend on each other in the base classes. We gave one example of this problem in a Java program in Figure 1. In this section, we rephrase this problem in the EO programming language.

EO is an untyped programming language, however, when using such a language a programmer still relies on some assumptions about inputs. For example, consider the following EO program, representing a function for computing a harmonic mean of two numbers:

```
[x y] > harmonic_mean
  2.mul ($.x.mul $.y) > product
  $.x.add $.y > sum
  $.product.div $.sum > @
```

Here, we assume that `x` and `y` void attributes of `harmonic_mean` are numeric objects (possessing attributes `add` and `mul` with some intuitive behaviour) and, moreover, $x + y \neq 0$.

These assumptions, or restrictions, on the void attributes (interpreted as arguments of a function) are the properties (preconditions) of the object `harmonic_mean` (interpreted as a function).

In this section, we will simply assume that properties of a method are represented by a logic predicate over the values of its arguments (void attributes, except `self`). However, it is important to note that it is often impractical to have precise properties of methods. Instead we will be interested in the following approximations:

Definition 3.1. Let `f` be a method with owner object referenced by the locator ℓ, and let x_1, …, x_n be formal arguments (void attributes) of `f`. We say that a logic predicate $P(x_1, \dots, x_n)$ is **over-approximating the properties** of `f`, if for all EO terms t_1, …, t_n satisfying $P(t_1, \dots, t_n)$ the following method call computes successfully: `ℓ.self.f ℓ.self t_1 . . . t_n`.

Example 3.2. Consider the following method:

```
[self x] > square_root
  $.x.sqrt > @
```

Predicate $P(x) = x > 10$ is over-approximating properties of `square_root`.

Example 3.3. A constant predicate $P(x_1, \dots, x_n) = \text{false}$ is over-approximating properties of any method.

Definition 3.4. Let `f` be a method with owner object referenced by the locator ℓ, and let x_1, …, x_n be formal arguments (void attributes) of `f`. We say that a logic predicate $P(x_1, \dots, x_n)$ is **under-approximating the properties** of `f`, if for all EO terms t_1, …, t_n such that the method call `ℓ.self.f ℓ.self t_1 . . . t_n` computes successfully, we have $P(t_1, \dots, t_n)$.

Example 3.5. Predicate $P(x) = x > -10$ is under-approximating properties of the method `square_root`, defined in the example 3.2.

Example 3.6. A constant predicate $P(x_1, \dots, x_n) = \text{true}$ is under-approximating properties of any method.

Now, to reformulate the unjustified assumptions problem in EO, we look at the properties of methods in decorated objects. In particular, consider EO program in Figure 2, that is analogous to the Java example in Figure 1. Computing `b.h b t` for some EO term `t` would result in the following evaluation:

```
b.h b t
  ⟶ b.g b t
  ⟶ b.@.g b t
  ⟶ a.g b t
  ⟶ b.f b (t.sub 1)
  ⟶ (t.sub 1).mul (t.sub 1)
```

```
[] > a
  [self x] > f
    $.x.sqrt > @
  [self y] > g
    $.self.f $.self ($.y.sub 1) > @
  [self z] > h
    $.z > @

[] > b
  ^.a > @
  [self x] > f
    $.x.mul $.x > @
  [self z] > h
    $.self.g $.self $.z > @
```

Figure 2: Unjustified assumptions in decorated objects in EO.

Note that here `t` is expected to be a numeric object, but there are no restrictions on its numeric value.

When we consider object `a` in isolation, we see that method `a.g` is referring to method `self.f`. Assuming object `a` is passed as `self`, we can refactor the definition of object `a` to this:

```
[] > a
  [self x] > f
    $.x.sqrt > @
  [self y] > g
    ($.y.sub 1).sqrt > @
  [self z] > h
    $.z > @
```

Note that, assuming only `a` can be passed as `self`, observational properties of the object `a` do not change. However, once we consider objects that decorate `a` as candidates for `self`, situation changes. Indeed, after this refactoring, calling `b.h b t` will result in a different evaluation process:

```
b.h b t
⟶ b.g b t
⟶ b.@.g b t
⟶ a.g b t
⟶ (t.sub 1).sqrt
```

Here we see that not only the behaviour changed, but now the numerical value of `t` is expected to be at least 1 (otherwise square root is undefined).

Definition 3.7. Let P be an EO program with top-level objects `a` and `b`, such that `b` decorates `a`. Then `b` is said to have an **unjustified assumptions defect** if there exist a n-ary method `b.f` and EO terms $\mathtt{t}_1, \ldots, \mathtt{t}_n$ such that $\mathtt{b.f\ b\ t_1\ \ldots\ t_n}$ computes successfully in P, but diverges when P is refactored by inlining at least one method in `a`.

4 DETECTING UNJUSTIFIED ASSUMPTIONS

To detect unjustified assumptions in decorator objects, we look at their properties before and after refactoring of the objects they decorate. This boils down to performing the inlining transformation on a decorated object, and comparing properties of decorator object's method before and after the transformation of the program. In this section, we specify the inlining algorithm in detail, present a possible approach to infer properties for methods, discuss what it means to compare properties and under which circumstances we can declare that a defect has been detected.

4.1 Inlining local methods

In general, direct inlining of a virtual method call does not preserve the semantics of a program. Indeed, a subclass may override the definition of the called method, invalidating the inlining. However, such behaviour might be undesired from a software developer's point of view. Mikhajlov and Sekerinski [6] discuss a several classes of problems, associated with unexpected mechanics of dynamic dispatch.

In this paper, we treat method inlining as a refactoring tool that, in most cases, is assumed to preserve the semantics of a program, assuming it does not have defects. In other words, we use inlining to compare supposedly equivalent programs — before and after inlining.

Considering a developer's perspective, inlining makes sense for method calls that are "close" to the place where the method itself has been defined. Essentially, we only consider local inlining of methods in a single object:

Definition 4.1. A method call $\ell\mathtt{.self.f}\quad \ell\mathtt{.self}\quad \mathtt{t}_1\ \ldots\ \mathtt{t}_n$ is **inlinable** in method `g` if the owner object of `g` is also an owner object of method `f` with arity n.

Example 4.2. Consider the EO program in Figure 2. Here, we have the following method calls:

(1) `$.self.f $.self ($.y.sub 1)` is an inlinable method call, since method `f` is defined in the same object that is owner of this method call;
(2) `$.self.g $.self $.z` is not an inlinable method call, since definition of method `g` is outside of the owner of the method call.

Intuitively, inlining a method call should be straightforward as we extract the `@` attribute of the method and perform substitution of formal arguments with actual terms passed to the method. However, matters are slightly more complicated in presence of local definitions. In particular, local definitions should be somehow transferred to the call site. Moreover, in transferred objects, locators should be adjusted, so that references they make stay the same. For this reason, we introduce locator increment operation on EO terms:

Definition 4.3. Let `t` be an EO term. Its subterm locator ℓ is said to be *open*, if ℓ references term outside of `t`. $\mathtt{t}\!\uparrow$ can obtained from `t` by changing all of its open locators ℓ to

(1) `^`, if $\ell \equiv$ `$`,
(2) $\mathtt{\hat{}.}\ell$, if $\ell \equiv$ `^.....^`.

Definition 4.4. Let `t` be an EO object term with methods `f` and `g`. **Inlining method calls to `g` in `f`** is a process of replacing all inlinable method calls to `g` in the object term attached to `f` according to the following rules:

(1) the object-containter of local definitions is generated in the same scope as the method call and attached to an attribute with any name, that does not introduce name clashes (for example, `args_g`). This object contains local definitions of `g`, where
   (a) void attributes of `g` are substituted with arguments of the method call with incremented locators: for example, $\mathtt{\$.x}_i$ is replaced by $\mathtt{t}_i\!\uparrow$;
   (b) attached attributes are left as they are.
(2) the method call $\ell\mathtt{.self}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_n$ is replaced by a term attached to `g.@`, where
   (a) void attributes of `g` are replaced by the arguments of method calls $\mathtt{t}_i$;
   (b) attached attributes of `g` become prefixed with the name of object-container, for example `$.b` is substituted by `$.args_g.b`.

Example 4.5. Consider the following EO program:

```
[] > obj
  [self x y] > g
```

```
    $.x.add $.y > sum
    $x.div $.sum > @
  [self a] > f
    100.sub $.a > b
    $.self.g $.self $.a $.b > @
```

After inlining call to `g` in `f`, object term `args_g` is introduced to the body of `f`, which contains attached attributes of `g` with necessary substitutions. Finally, the method call is replaced by its return value with corresponding substitutions:

```
  [self a] > f
    100.sub $.a > b
    [] > args_g
      ^.a.add ^.b > sum
    $.a.div $.args_g.sum > @
```

4.2 Property inference for methods

To analyze the properties of methods in EO, we rely on a simple abstract interpretation of EO terms. In particular, we assign to each EO term an optional value expression and a logical formula. A value expression, if present, specifies the value of a term in some domain. A value expression may depend on values of attributes of external objects that act similar to free variables. A logic formula specifies restrictions of the context — values of attributes of external objects.

Example 4.6. Consider the following EO program:

```
[self x] > recip
  1.div $.x > @
```

Here, the value expression for the term `1.div $.x` is $1/v_x$ whenever v_x is a value of `$.x`. At the same time, the logical formula associated with this term is $(v_x \neq 0)$.

We limit the scope of this paper, to consider simple numeric operations, booleans and basic control primitives of EO programming language. We will use the following notation:

Definition 4.7. Syntax for **value expressions** and **logical formulae** is given in Figure 3. Let `term` be an EO term, E be a value expression, and P be a logic formulae. A three-way relationship between EO terms, value expressions, and properties, called **interpretation judgement**, is defined by rules in Figure 4. We write $\left\{ \texttt{term} \equiv E \middle| P \right\}$ for an interpretation judgement, meaning that value expression for EO term `term` is E under assumption that outer object attributes satisfy P.

Many inference rules in Figure 4 are quite straightforward. For example, the rule for $\texttt{t}_1\texttt{.add t}_2$ says that as long as we can interpret $\texttt{t}_1$ with value e_1 and properties p_1, and $\texttt{t}_2$ with value e_2 and properties p_2, we can interpret $\texttt{t}_1\texttt{.add t}_2$ with value $e_1 + e_2$ and properties $p_1 \wedge p_2$.

Attribute terms are converted into variables. Importantly, different attribute terms can reference the same value in an EO program. To make sure, all these terms are normalized, the object rule in Figure 4 relabels free variables correspondingly.

Example 4.8. Consider the following EO program:

```
[self x] > recip
  1.div $.x > @
```

$$
\begin{aligned}
E ::= {} & \mathsf{true} \mid \mathsf{false} \mid \mathsf{num\_const} \\
& \mid E_1 + E_2 \mid E_1 * E_2 \mid E_1^{E_2} \\
& \mid E_1 < E_2 \mid E_1 \leq E_2 \mid E_1 = E_2 \mid E_1 \neq E_2 \\
& \mid \neg E \mid E_1 \wedge E_2 \mid E_1 \vee E_2 \mid E_1 \implies E_2 \\
& \mid e_{\texttt{ℓ.x}} \qquad \text{(outer object attribute value)} \\
P ::= {} & \mathsf{true} \mid \mathsf{false} \\
& \mid E_1 < E_2 \mid E_1 \leq E_2 \mid E_1 = E_2 \mid E_1 \neq E_2 \\
& \mid \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \\
& \mid p_{\texttt{ℓ.x}} \qquad \text{(outer object attribute property)}
\end{aligned}
$$

Figure 3: Syntax for value expressions and logical formulae.

We have the following judgement:

$$\left\{ \texttt{1.div \$.x} \equiv 1/V_{\texttt{\$.x}} \middle| V_{\texttt{\$.x}} \neq 0 \right\}$$

4.3 Detecting problematic decoration

With inlining and property inference, we are ready to detect unjustified assumptions in decorator objects. The idea is straightforward: given a decorator object, we inline methods in one or all of the objects it decorates, and see how inferred properties changed for the decorator object. We assume that inlining methods in any object `x` should not break observational behaviour (introduce errors) in any other object `y` that decorates, perhaps, indirectly, object `x`.

We limit the scope of the analysis to the methods of objects. For any method `f` of an object `x`, we analyse inferred properties before and after inlining some methods in a given program, yielding two logical predicates: $p^{\text{before}}_{\texttt{x.f}}(x_1, \ldots, x_n)$ and $p^{\text{after}}_{\texttt{x.f}}(x_1, \ldots, x_n)$. Now, we want to know whether inputs that worked before, continue working after refactoring. Thus, we are interested in the value of the following logical formula:

$$\forall x_1, \ldots, x_n . p^{\text{before}}_{\texttt{x.f}}(x_1, \ldots, x_n) \implies p^{\text{after}}_{\texttt{x.f}}(x_1, \ldots, x_n) \qquad (1)$$

Intuitively, when this formula is true, then we interpret it as indicating that there is no defect detected. When it is false, then it means there exist some inputs $x_1, \ldots, x_n$ such that they worked before the inlining, and stopped working after.

Note that, in general, predicates $p^{\text{before}}_{\texttt{x.f}}$ and $p^{\text{after}}_{\texttt{x.f}}$ will not reflect the properties exactly, and instead will somehow approximate them. Thus, it is important to understand how well does Equation 1 approximate the presence of a defect in a program.

Proposition 4.9. *Let* $p^{\text{before}}_{\texttt{x.f}}$ *and* $p^{\text{after}}_{\texttt{x.f}}$ *be logical predicates approximating properties of method* `x.f` *before and after a revision of the program. Then, we can distinguish two important cases:*

(1) *If* $p^{\text{before}}_{\texttt{x.f}}$ *is an over-approximation and* $p^{\text{after}}_{\texttt{x.f}}$ *is an under-approximation, then Equation 1 is* ***sound*** *in the sense that if its value is* false*, then the corresponding EO program contains*

$$\frac{\texttt{lit} \text{ is a literal representing constant } c}{\left\{\texttt{lit} \equiv c \,\middle|\, \mathsf{true}\right\}}\ \text{literal}$$

$$\frac{\left\{\texttt{t}_1 \equiv e_1 \,\middle|\, p_1\right\} \qquad \left\{\texttt{t}_2 \equiv e_2 \,\middle|\, p_2\right\}}{\left\{\texttt{t}_1\texttt{.add t}_2 \equiv e_1 + e_2 \,\middle|\, p_1 \land p_2\right\}}\ \text{addition} \qquad \frac{\left\{\texttt{t}_1 \equiv e_1 \,\middle|\, p_1\right\} \qquad \left\{\texttt{t}_2 \equiv e_2 \,\middle|\, p_2\right\}}{\left\{\texttt{t}_1\texttt{.div t}_2 \equiv e_1/e_2 \,\middle|\, p_1 \land p_2 \land (e_2 \neq 0)\right\}}\ \text{division}$$

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

$$\frac{\left\{\texttt{t}_1 \equiv e_1 \,\middle|\, p_1\right\} \qquad \left\{\texttt{t}_2 \equiv e_2 \,\middle|\, p_2\right\} \qquad \left\{\texttt{t}_3 \equiv e_3 \,\middle|\, p_3\right\}}{\left\{\texttt{t}_1\texttt{.if t}_2\ \texttt{t}_2 \equiv \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \,\middle|\, p_1 \land ((e_1 \lor p_2) \lor (\neg e_1 \lor p_3))\right\}}\ \text{if}$$

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

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

$$\frac{\forall i \in \{1, \ldots, m\}, \left\{\texttt{e}_i \equiv e_i \,\middle|\, \exists z^i_1, \ldots, z^i_{n_i}.p_i\right\} \qquad \left\{\texttt{e}_@ \equiv e_\varphi \,\middle|\, \exists z^\varphi_1, \ldots, z^\varphi_{n_\varphi}.p_\varphi\right\}}{\left\{\begin{array}{l}\texttt{[]} \\ \quad \texttt{e}_1 > \texttt{y}_1 \\ \quad \ldots \\ \quad \texttt{e}_m > \texttt{y}_m \\ \quad \texttt{e}_@ > @\end{array} \equiv e_\varphi[e_{\texttt{\$.y}_i} \mapsto e_i] \,\middle|\, \exists(e_{\texttt{\$.y}_1}, \ldots, e_{\texttt{\$.y}_m}, z^i_1, \ldots, z^i_{n_i}, \ldots, z^\varphi_1, \ldots, z^\varphi_{n_\varphi}).p_1 \land \ldots \land p_n \land p_\varphi \land e_{\texttt{\$.y}_i} = e_i\right\}}\ \text{object}$$

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

Figure 4: Inference rules for value expressions and properties of EO terms.

some unjustified assumptions defect. Moreover, it is method `x.f` *that relies on those unjustified assumptions.*

(2) *If* $p^{\text{before}}_{\texttt{x.f}}$ *is an under-approximation and* $p^{\text{after}}_{\texttt{x.f}}$ *is an over-approximation, then Equation 1 is* ***complete*** *in the sense that if the corresponding versions of EO program change the properties of* `x.f`*, then the value of the formula is* false.

In static analysis, soundness of a tool is typically preferred over completeness. Thus, we would like to perform analysis by inferring over-approximated properties before inlining and under-approximated properties after inlining. Unfortunately, this is not always practical. Consider the following method:

```
[self x] > f
  seq > @
    debug.print x
    x
```

Here, it is possible that the static analyzer is unaware of `debug.print`, so it can only over-approximate it with $P_{\texttt{debug.print}}(x) = \text{false}$. This would, in turn, make it seem that method `f` itself cannot accept any inputs. While technically, this is a valid over-approximation, it is not useful. Thus, in practice we are typically ignoring unknown definitions in the program (by under-approximating them with a constant true predicate).

5 IMPLEMENTATION IN SCALA

We have implemented the approach described in Section 4 in Scala programming language. In particular, we have followed functional programming approach for abstract syntax processing and analysis.

The abstract syntax for EO has been described using Scala's **case**-classes, following the approach similar to the one described by Kubuszok [5]. For our analysis, we are interested primarily in EO expressions. While in the previous sections, we assumed only one form of application (nameless application), technically EO allows for named applications as well. Note that here, and in the previous sections we do not attempt to analyze partial application (i.e. applications that result in objects with void attributes). Similarly, we are not analyzing the variable argument (vararg).

Expressions in EO can be bound to names or live on their own. To this end, we declare the following types of bindings:

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

sealed trait EOBnd[+A] { val expr: A }

```

```
sealed case class EOAnonExpr[+A](
  override val expr: A,
) extends EOBnd[A]

sealed case class EOBndExpr[+A](
  bndName: Name,
  override val expr: A,
) extends EOBnd[A]
```

Here, the type parameter **A** is a placeholder for the actual type of expressions. We later use a fixpoint constructor **Fix** to generate the recursive type of EO expressions **EOExprOnly**.

The types for the abstract syntax tree nodes are defined as follows:

```
// the common ancestor of expression node classes
sealed trait EOExpr[+A]

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

// access to an attribute of a locator
sealed case class EODotLocator[+A](
  name: String,
  locator: BigInt
) extends EOExpr[A]

// access to an attribute of another expression
sealed case class EODot[+A](
  src: A,
  name: String
) extends EOExpr[A]

// application of an object to a list of other expressions
sealed case class EOCopy[+A](
  trg: A,
  args: NonEmptyVector[EOBnd[A]]
) extends EOExpr[A]

// literals
sealed trait EOData[+A] extends EOExpr[A]
sealed case class EOIntData[+A](int: Int) extends EOData[A]
sealed case class EOBoolData[+A](bool: Boolean) extends EOData[A]
// there are more primitive literals
```

Finally, we tie the knot, defining the recursive type of expressions:

```
type EOExprOnly = Fix[EOExpr]
```

Note that before we begin processing the abstract syntax tree, we restore all omitted locators as this simplifies analysis.

Inlining is performed in a straightforward manner, following rules from Definition 4.4. We choose to inline all inlinable methods in the entire EO program at once:

```
def inlineAllCalls(prog: EOExprOnly): EOExprOnly
```

5.1 Property inference

To compute values of logical formulae, we choose to rely on scala-smtlib, a lightweight abstraction over SMT-LIB [1] with Princess [7] backend. To build the formulae we implement a recursive algorithm that traverses the AST, accumulating the following information:

```
final case class Info(
  forall: List[SortedVar],
  exists: List[SortedVar],
  value: Term,
  properties: Term
) {}
```

We use `forall` to collect a list of void attributes used in method's body. Variables corresponding to local definitions, including nested definitions in locally defined objects, are collected in `exists`. Value expression and inferred properties are stored in `value` and `properties` correspondingly. A valid information structure for a method has `value` and `properties` that rely only on variables that are defined in its `exists` and `forall` fields.

Assuming we have information about defined and available methods, we define the following function to extract information structure from a given EO expression:

```
def extractInfo(
    depth: List[String],
    expr: EOExprOnly,
    availableMethods: Set[Name]
  ): Info
```

Here, `depth` is used to track the nested structure of definitions, to interpret equivalent locators in the same way.

6 CONCLUSION

We have presented the problem of unjustified assumptions in decorated objects, a reformulation of a similar problem for subclasses. We have suggested an approach for detecting such problems in EO code, based on inlining and property inference. We have shown that our approach to detection is sound — a successful detection means that the problem is present in the original code, assuming an interpretation of methods in EO programs as virtual methods, and no unknown primitives are used. In addition to that, if EO program is a result of a faithful translation from another object-oriented programming language, the detection remains sound for the program in the original language.

We have discussed our implementation of the approach using Scala programming language, in combination with Princess SMT solver. Although in presented work, we have focused on basic properties, we believe that current work can be extended in a straightforward manner, to include more primitives. Properly supporting recursion, however, may require revision of the approach, and we leave this for future work.

We have presented the combined technique of inlining and property inference specifically for the detection of the unjustified assumptions. However, we think that these techniques can be used for detection of other important problems, such as verifying whether Liskov substitution principle applies to certain subclasses.

We also note that property inference, as presented in this work, can be related to type inference for some type system, imposed on top of untyped EO programming language. In particular, we think it is important for future static analysis of EO programs, to be able to infer information about objects, such as possible lists of their attributes, together with types of those attributes.

REFERENCES

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.
[2] Joshua Bloch. 2018. *Effective Java* (3 ed.). Addison-Wesley, Boston, MA. https://www.safaribooksonline.com/library/view/effective-java-third/9780134686097/
[3] Yegor Bugayenko. 2021. EOLANG and phi-calculus. arXiv:2111.13384 [cs.PL]
[4] Yegor Bugayenko. 2021. Reducing Programs to Objects. arXiv:2112.11988 [cs.PL]
[5] Mateusz Kubuszok. 2019. *AST playground: recursion schemes and recursive data.* https://kubuszok.com/2019/ast-playground-recursion-schemes-and-recursive-data/
[6] Leonid Mikhajlov and Emil Sekerinski. 1998. A Study of The Fragile Base Class Problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECCOP '98)*. Springer-Verlag, Berlin, Heidelberg, 355–382.
[7] Philipp Rümmer. 2008. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LNCS, Vol. 5330)*. Springer, 274–289.
[8] C. Szyperski, D. Gruntz, and S. Murer. 2002. *Component Software: Beyond Object-oriented Programming*. ACM Press. https://books.google.ru/books?id=U896iwmtiagC