Автономная некоммерческая организация высшего образования
«Университет Иннополис»

АННОТАЦИЯ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
(БАКАЛАВРСКУЮ РАБОТУ)
ПО НАПРАВЛЕНИЮ ПОДГОТОВКИ
09.03.01 – «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

НАПРАВЛЕННОСТЬ (ПРОФИЛЬ) ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ
«ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

Тема **Статический анализ программ на объектно-ориентированных языках программирования с использованием промежуточного представления, основанного на Elegant Objects**

Выполнил **Олокин Михаил Александрович**

подпись

Иннополис, Innopolis, 2022

Оглавление

Аннотация

В данной диссертации описывается пробная реализация статического анализатора для объектно-ориентированных программ, которые обнаруживают два дефекта семейства ”хрупких базовых классов непредвиденную взаимную рекурсию и неоправданное допущение в модификаторе (подклассе). Реализация анализаторов опирается на промежуточное представление, называемое EO (сокращение от Elegant Objects), которое основано на φ-исчислении - формализации общей семантики объектно-ориентированных языков программирования, вдохновленной шаблоном проектирования ”декоратор”. Правильность анализаторов была проверена с использованием подхода тестирования на основе свойств, а также рукописных модульных тестов для выявления важных краевых случаев.

Глава 1

Введение

За последние два десятилетия языки объектно-ориентированного программирования получили широкое распространение. По состоянию на март 2022 года пять первых мест в индексе TIOBE [1] занимают Python, C, Java, C++ и C#. Четыре из этих языков (за исключением C) считаются объектно-ориентированными и, как следует из индекса, широко распространены и используются в крупномасштабных коммерческих продуктах.

1.1 Объектно-ориентированные языки программирования

Согласно [1], *объектно-ориентированные* языки программирования - это языки, в которых основной единицей абстракции является *объект*. Объекты инкапсулируют *данные*, которые являются значениями некоторого типа. Некоторые языки, например, Java и C++, различают *примитивные типы*, которые представляют низкоуровневые конструкции, такие как числа или буле-

[1] https://www.tiobe.com/tiobe-index/

вы значения, и *объектные типы*, которые представляют составной тип. Объекты могут также содержать операции над указанными данными, известные как *методы*. Методы могут принимать параметры и возвращать значение.

Объекты также должны подчиняться некоторым определенным свойствам. Как утверждает [1], ”степень, в которой конкретный язык удовлетворяет этим свойствам, определяет, насколько он является объектно-ориентированным языком”. Этими свойствами являются:

- Инкапсуляция - объект должен представлять четко определенный интерфейс, через который он должен потребляться. Несущественные детали того, как объект реализует этот интерфейс, должны быть *скрыты* от потребителя.

- Наследование - это механизм, с помощью которого объекты могут совместно использовать функциональность и расширять поведение других объектов. Наследование - сложный механизм, и его реализация отличается в разных языках. Согласно [1], ”Наследование позволяет программистам повторно использовать определения ранее определенных структур. Это явно сокращает объем работы, требуемой при создании”.

- Полиморфизм - возможность определять операции над объектами таким образом, что они могут принимать и возвращать значения нескольких типов.

- Динамическое (или позднее [2]) связывание - реализация метода, который будет выполняться на объекте, выбирается во время выполнения. Это означает, что реализация, которая используется во время выполнения программы, может быть *различной* от реализации типа, который известен статически (т.е. во время компиляции).

1.2 Критика

Вместе с растущим распространением, методы и языки ОО-программирования получили значительное количество обоснованной критики. Мэнсфилд [3] упоминает большинство из этих жалоб, в конечном итоге утверждая, что ”...с ООП-инфлектированными языками программирования, компьютерное программное обеспечение становится более многословным, менее читаемым, менее описательным, его труднее модифицировать и поддерживать”. Многие из этих критических замечаний превращаются в рекомендации, такие как знаменитый документ ”Design patterns: elements of reusable object-oriented software”[4]. Однако такие рекомендации не являются частью спецификации языка и поэтому не могут быть реализованы компилятором языка. Это приводит к тому, что эти рекомендации часто неправильно интерпретируются или используются слишком часто, особенно новичками.

1.3 Инструменты анализа

. Чтобы смягчить эту сложность и обеспечить применение хороших практик, разработчики создали множество программных инструментов. Эти инструменты можно разделить на две категории: **динамические** анализаторы и **статические** анализаторы. **Динамические** анализаторы (также известные как *профайлеры*) проверяют состояние программы в процессе ее выполнения. Динамические анализаторы собирают важную информацию о выполнении программы, такую как загрузка процессора и потребление памяти, и представляют ее в удобочитаемой форме. Эта информация очень важна в приложениях, где производительность играет важную роль. К сожалению,

эти инструменты требуют выполнения анализируемой программы, что может быть дорого или даже невозможно, например, когда программа должна выполняться на специализированном оборудовании.

Напротив, **статические** анализаторы проверяют исходный код программы (или одно из его промежуточных представлений) *без его выполнения*, чтобы найти общие ошибки, анти-паттерны и отклонения от принятых стилевых конвенций. Выполнение таких инструментов обычно не отнимает много времени и не требует больших затрат, поэтому они являются важной частью конвейеров непрерывной интеграции [5] и интегрированных сред разработки (IDE). Несмотря на склонность к ложным срабатываниям, инструменты статического анализа могут с большей точностью определить место ошибки.

В отличие от динамических анализаторов, статические анализаторы работают с исходным кодом, что позволяет им проверять программу с точки зрения более высокого уровня. Это означает, что статические анализаторы могут улучшить отчетность об ошибках компиляторов языков программирования, обнаружить больше проблем и даже автоматически их устранить.

Перед анализом многие инструменты статического анализа преобразуют исходный текст целевого языка в некоторое промежуточное представление. Это делается по нескольким причинам. Как правило, это делается для того, чтобы извлечь из исходного кода информацию, необходимую для анализа. Другим распространенным случаем использования промежуточного представления является следующее заставить статический анализатор работать с более чем одним целевым языком. В этом случае представление служит общей основой для различных анализаторов. Примерами промежуточного представления являются LLVM [6] и Jimple [7] (используется в SOOT [8]).

1.4 Цель исследования

В данной диссертации мы представляем реализацию модуля для статического анализатора объектно-ориентированных программ, который принимает на вход представление программы на языке Elegant Objects (EO)[9] и выдает на выходе простые сообщения об ошибках. EO - это промежуточное представление, основанное на ϕ-calculus, формальной модели, которая предназначена для унификации различной семантики объектно-ориентированных языков. Он также претендует на звание языка с минимальной многословностью, предоставляя минимально необходимый набор операций. Сочетание строгой формальной основы и сокращенного набора функций делает EO мощным промежуточным представлением для статического анализатора, который должен быть способен отлавливать многие ошибки, характерные для ОО-программ. В диссертации описана пробная реализация обнаружения двух дефектов семейства ”хрупких базовых классов”[10]: ”непредвиденная взаимная рекурсия”и ”неоправданное допущение в модификаторе”.

Остальная часть диссертации построена следующим образом: в главе 2 рассматриваются существующие работы по поиску ошибок в ОО программах, в главе 3 описывается семантика ОО и то, как она может представлять объектно-ориентированные программы, в главе 4 описывается реализация анализатора.Часть 5 завершает диссертацию.

Глава 2

Обзор литературы

В этой главе представлен обзор теоретических концепций, на которые опирается реализация. Раздел 2.2 кратко описывает соответствующие части φ-исчисления: его синтаксис и семантику. Раздел 2.3 объясняет, как φ-исчисление отображается на EO, промежуточное представление, на котором работают анализаторы. Раздел 2.4 показывает, как кодировать основные объектно-ориентированные конструкции (классы, методы, наследование) с помощью EO.

2.1 Метод отбора статей

. Работы, связанные с φ-исчислением и EO, появились сравнительно недавно и в основном не опубликованы. Те работы, которые уже опубликованы, были предоставлены научным руководителем. Препринты неопубликованных работ были любезно предоставлены авторами.

2.2 φ-исчисление

EO - это язык программирования, реализующий φ-calculus, формальную модель для объектно-ориентированных языков программирования, первоначально представленную Бугаенко [9]. В этой диссертации мы используем уточнение φ-calculus, предложенное Кудасовым и Сим [11].

2.2.1 Объекты и атрибуты

В основе φ-исчисления лежит понятие **объект**.

Definition 1 (Объекты и атрибуты)**.** . **Объект** - это набор пар $[\![n_0 \mapsto o_0, n_1 \mapsto o_1, \ldots, n_i \mapsto o_i, \ldots]\!]$, где n_i - уникальный идентификатор, а o_i - объект. Такие пары известны как **атрибуты**. Первый элемент - это **имя атрибута**, второй элемент - **значение атрибута**. Пустой набор $[\![]\!]$ также является допустимым объектом. Атрибут, в котором вторым элементом является $[\![]\!]$, называется **void** или **free**. В противном случае он называется **attached**.

Атрибуты объекта могут быть доступны по их именам через точечную нотацию:

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

В данном случае, это сводится к объекту y, который определен в другом месте. $\rightsquigarrow$ означает ”сводится к”или ”вычисляется в”.

2.2.2 Аппликация

Аппликация может быть использована для создания нового объекта, в котором заданы значения некоторых или всех свободных атрибутов. Други-

ми словами, применение может быть использовано для создания *закрытых* объектов из *абстрактных* объектов.

Definition 2 (Абстрактные и закрытые объекты)**.** . Если объект имеет один или несколько свободных атрибутов, он называется **abstract** или **open**. В противном случае, он называется **закрытый**.

Например, объект a в 2.2 соответствует точке в двумерном пространстве с координатами $x = 1, y = 2$. Объекты 1 и 2 могут быть определены в терминах φ-исчисления, однако само определение выходит за рамки данной диссертации.

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

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

$$\tag{2.3}$$

$$a \rightsquigarrow [\![x \mapsto 1, y \mapsto 2]\!] \tag{2.4}$$

2.2.3 Локаторы

Пересмотр φ-исчисления Кудасовым и Сим [11] также определяет специальные объекты, называемые **локаторами**, которые обозначаются как ρ^i, где $i \in \mathbb{N}$. Локаторы позволяют объектам ссылаться на другие объекты относительно объекта, в котором используется локатор. Например, это может быть использовано для (но не ограничивается этим) кодирования определения атрибутов в терминах других атрибутов данного объекта. Предположим,

есть объект x:

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

Выражение $x.a$ будет сведено к значению объекта c. Это происходит потому, что $x.a$ ссылается на $x.b$ через ρ^0, что означает непосредственный объемлющий объект. В более сложных примерах, таких как 2.5.

$$x := [\![a \mapsto [\![c \mapsto \rho^1.b]\!], b \mapsto d]\!] \tag{2.5}$$

$$x.a.c \rightsquigarrow d \tag{2.6}$$

ρ можно использовать для определения атрибутов внутренних объектов в терминах атрибутов внешних объектов, или даже самих внешних объектов.

2.2.4 φ-атрибут

Объекты могут определять специальный атрибут с именем φ. Этот атрибут перенаправляет доступ к атрибуту на его значение, если у окружающего объекта нет атрибута с таким именем (рис. 2.7).

$$a := [\![d \mapsto y]\!] \tag{2.7}$$

$$x := [\![\varphi \mapsto a, c \mapsto g]\!] \tag{2.8}$$

$$x.d \rightsquigarrow x.\varphi.d \rightsquigarrow y \tag{2.9}$$

Если атрибут присутствует и в объекте, и в его φ-атрибуте, приоритет

$$
\begin{array}{l}
\mathit{fib} := [\![\\
\quad n \mapsto [\![]\!], \\
\quad \varphi \mapsto \rho^0.n.\mathit{less}(n \mapsto 2).\mathit{if}(\\
\quad\quad \mathit{ifTrue} \mapsto n, \\
\quad\quad \mathit{ifFalse} \mapsto \\
\quad\quad \mathit{fib}(n \mapsto \rho^0.n.\mathit{sub}(n \mapsto 1)) \\
\quad\quad .\mathit{add}(n \mapsto \mathit{fib}(n \mapsto \rho^0.n.\mathit{sub}(n \mapsto 2)) \\
\quad) \\
\quad) \\
\quad) \\
\quad]\!]
\end{array}
$$

Рис. 2.1: Fibonacci numbers in φ-calculus

имеет атрибут в объекте:

$$a := [\![d \mapsto y]\!]$$

$$x := [\![\varphi \mapsto a, \mathbf{d} \mapsto g]\!]$$

$$x.d \rightsquigarrow g$$

В 2.8, Бугаенко [9] называет объект a **декорированный объект**, где часть ”декорированный”относится к шаблону декоратора, описанному в [4, Глава 4]. Эта техника расширения объекта также известна как *делегирование*. [12] в объектно-ориентированных языках.

2.2.5 Сложный пример

. Связывая все воедино, рисунок 2.1 показывает, как φ-исчисление может быть использовано для вычисления чисел Фибоначчи.

2.3 EO

EOLANG, или просто EO, - это язык программирования, созданный Бугаенко [9], который является прямой реализацией φ-calculus с некоторыми расширениями. Однако их реализация содержит особенности, которые не имеют отношения к теме данной диссертации. Более того, существует заметное различие между версией EO Бугаенко и ϕ-исчислением [11] в определении локаторов (или ”родительских объектов”). В работе Бугаенко локаторы являются *атрибутами*, тогда как в работе [11] они являются *объектами*. В данной диссертации, по аналогии с φ-исчислением, мы будем использовать другую версию EO, которая является прямым переводом исчисления, определенного в 2.2. Таблица перевода показана на рисунке 2.2.

2.4 Описание объектно-ориентированных программ с помощью EO

Прежде чем анализировать программы, написанные на объектно-ориентированных языках программирования, необходимо перевести их в EO, сохранив семантику исходного языка. В этом разделе представлена упрощенная версия такой кодировки, которая предполагается анализаторами, описанными в данной диссертации. Кодировка была в значительной степени заимствована из [13] с некоторыми изменениями, направленными в основном на упрощение процесса анализа.

	φ-исчисление	EO

Объекты	$obj := [\![a \mapsto x, b \mapsto y]\!]$	`[] > obj`
		`  x > a`
		`  y > b`

Свободные атрибуты	$point := [\![x \mapsto [\![]\!], y \mapsto [\![]\!]]\!]$	`[x y] > poin`
Аппликация	$a := point(x \mapsto 1, y \mapsto 2)$	`point 1 2 > a`

φ-атрибут	$x := [\![\varphi \mapsto a, c \mapsto g]\!]$	`[] > x`
		`  a > @`
		`  g > c`

Пример с числами Фибоначчи	Fig. 2.1	`[n] > fib`
		`  ($.n.less 2).if >`
		`  n`
		`  (fib ($.n.sub`
		`  (fib ($.n.sub`

ρ^0	ρ^0	$
ρ^1	ρ^1	^
ρ^3	ρ^3	^.^.^

Рис. 2.2: Отображение φ-исчисления в EO

2.4.1 Классы

Классы моделируются как закрытые объекты EO. Атрибуты уровня класса (т.е. ”статические”) становятся атрибутами объекта класса. Конструктор представлен атрибутом-объектом ”new”объекта класса. Этот объект может принимать параметры для создания экземпляра объекта.

Все атрибуты и методы экземпляра определяются внутри объекта, возвращаемого объектом ”new”. В EO наследование моделируется как украшение. Экземпляры классов (они же объекты в Java) создаются путем применения объекта ”new”к требуемым параметрам.

2.4.2 Методы

Методы моделируются как объекты EO, аналогично классам. Эти объекты могут принимать параметры. Методы экземпляра должны принимать специальный атрибут **self** в дополнение к другим параметрам. Этот параметр используется для передачи экземпляра объекта, вызывающего метод (отсюда и название - ”self”). Параметр ”self”можно использовать для вызова методов экземпляра внутри других методов экземпляра. Вызов метода принимает следующий вид:

```
self.method_name self arg1 arg2 итд ..
```

Возвращаемое значение метода представлено значением атрибута φ (символ ”@”в EO). Для того чтобы вызвать метод экземпляра, нам необходимо сначала инстанцировать объект. Затем мы можем вызвать метод, обратившись к атрибуту экземпляра с именем метода и передав ему объект экземпляра в качестве первого аргумента.

2.4.3 Примеры перевода

Примеры такого перевода, примененного к простым Java-программам, можно найти в последующих разделах, а именно на рисунках 2.3 и 2.5.

2.5 Проблема хрупкого базового класса

2.5.1 Непредсказуемая взаимная рекурсия

Непредвиденная взаимная рекурсия - это проблема, которая возникает в результате неограниченного наследования. Предположим, у нас есть объект Base с двумя методами - *f* и *g*. Метод *g* вызывает метод *f*, а *f* - нет.

Затем существует класс Derived, который расширяет Base и переопределяет метод *f* таким образом, что он вызывает *g*. Когда мы вызываем метод *f* на экземпляре Derived, мы получаем ошибку переполнения стека: метод *f* вызывает метод *g*, метод *g* вызывает метод *f* и так далее (Рис. 2.3).

Важно отметить, что мы не заинтересованы в обнаружении взаимной рекурсии между двумя методами одного класса. Нас интересуют только те случаи, когда взаимная рекурсия возникает в результате переопределения одного из методов суперкласса. Пример (Рис. 2.4) показывает класс с двумя взаимно рекурсивными методами *isOdd* и *isEven*. В данном случае рекурсия ожидаема и необходима, поэтому она не является дефектом.

2.5.2 Необоснованное предположение в подклассе

. Этот дефект [10, Раздел 3.3] возникает, когда суперкласс рефакторят путем *инлайнинга* вызовов метода, который может быть переопределен подклассом. Термин инлайнинг означает замену вызова метода его телом. Рас-

```
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

Рис. 2.3: Пример непредвиденной взаимной рекурсии

```
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

Рис. 2.4: Пример без непредвиденной взаимной рекурсии.

смотрим пример (Рис. 2.5). Класс *M* расширяет класс *C*, переопределяя метод *l*, чтобы ослабить его предусловие. Следовательно, предусловие в методе *m* класса *M* также ослаблено, потому что оно зависит от вызова метода *l*.

Теперь предположим, что класс *C* пришел из некоторой внешней библиотеки, а класс *M* определен в пользовательском коде. Сопровождающий библиотеки решает рефакторить класс *C*, инлайнируя вызов *l* в методе *m* (Рис. 2.6). Понаблюдайте, что происходит с классом *M*. Теперь, когда *m* в классе base имеет assert, переопределение метода *n* в классе *M* имеет усиленное предусловие по сравнению с его версией в классе *C*. Поэтому, казалось бы, безопасный рефакторинг в базовом классе нарушил инварианты в подклассах. Название дефекта происходит от того, что подклассы обычно *M предполагают*, что метод *m* должен быть реализован через метод *l*. Примеры на рис. 2.5 и 2.6 показывают, что такое предположение действительно не оправдано, и сопровождающие класса *C* могут изменить его по своему усмотрению.

```
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

Рис. 2.5: Пример необоснованного предположения в подклассе (до пересмотра)

```
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

Рис. 2.6: Пример необоснованного предположения в подклассе (после пересмотра)

Глава 3

Методология

В этой главе описывается организация процесса исследования и внедрения. Раздел 3.1 описывает процесс исследования, который предшествовал внедрению. Раздел 3.2 охватывает инструменты и технологии, использованные в проекте. Наконец, раздел 3.3 дает краткий обзор структуры модулей проекта.

3.1 Исследование проблемы

Прежде всего, мы изучили проблемы хрупких базовых классов, определив, какие из них будут наиболее подходящими для реализации. После того, как мы остановились на непредвиденной взаимной рекурсии и неоправданном допущении в подклассе, мы начали разрабатывать алгоритмы вместе с тестовыми примерами, которые использовались в основном для справки. Эти тестовые примеры позже стали частью окончательного набора тестов анализаторов. После того, как алгоритмы были доработаны и одобрены руководителем, мы приступили к реализации.

3.2 Разработка

Мы решили разместить репозиторий исходного кода нашего проекта на Github [1]. Это было сделано в основном потому, что команда уже была знакома с этой платформой, и все необходимые настройки требовали минимальных усилий. Еще одним аспектом Github, который привлек наше внимание, была функция под названием Github Actions [2]. Эта функция позволила легко разработать и интегрировать конвейеры непрерывной интеграции [5] и непрерывного развертывания [14] в наш репозиторий без необходимости использования самостоятельных хостинговых решений. Эти конвейеры были сконфигурированы таким образом, чтобы они запускались при каждом отправлении в мастер-ветку, отклоняя отправку, если исходный код не прошел тесты, компиляцию или линтинг. Такая конфигурация гарантирует, что код в мастер-ветке будет соответствовать стандартам качества в любой момент времени в процессе разработки.

Реализация анализаторов была полностью выполнена на **Scala** [3] - современном языке программирования с поддержкой высокоуровневых концепций, таких как структурное сопоставление шаблонов [15] и алгебраические типы данных [16]. Scala компилируется в байт-код виртуальной машины Java (JVM). Это позволяет использовать реализацию в качестве библиотеки в любом другом проекте, совместимом с JVM, будь то Scala или Java. Кроме того, программы, скомпилированные в байт-код JVM, могут быть запущены без изменений на любом устройстве, на котором может работать Java Virtual Machine.

[1] `https://github.com/`

[2] `https://github.com/features/actions`

[3] `https://www.scala-lang.org/`

В проекте используется инструмент сборки под названием **sbt** [4], который позволяет компилировать несколько модулей Scala одновременно. Отличительной особенностью **sbt** является возможность кросс-компиляции кода Scala, благодаря чему он совместим со многими версиями Scala и Java. Он также поддерживает множество плагинов, которые улучшают процесс разработки. Мы использовали два таких плагина: **scalafmt** [5], автоматический форматтер исходного кода, и **scalafix** [6] , линтер и анализатор кода с поддержкой рефакторинга в рамках всего проекта.

Анализаторы публикуются в виде **JAR** [7] и могут быть загружены из репозитория **Maven Central** [8].

Исходный код проекта доступен на **Github** [9]. Там же можно найти инструкции по запуску и внесению вклада в проект.

3.3 Структура модуля

Исходный код анализаторов был разделен на несколько модулей:

- Core, который содержит определение для EO AST (Abstract Syntax Tree). Это AST используется в качестве входных данных для всех алгоритмов анализа.
- Analyses, которая содержит реализации анализаторов.
- Backends, которая содержит алгоритмы, преобразующие EO AST во что-то другое. Пока что единственным бэкендом является бэкенд обыч-

[4] https://www.scala-sbt.org/
[5] https://scalameta.org/scalafmt/
[6] https://scalacenter.github.io/scalafix/
[7] https://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html
[8] https://search.maven.org/search?q=g:org.polystat.odin
[9] https://github.com/polystat/odin

ного текста: он преобразует EO AST в его синтаксически корректный эквивалент в исходном коде EO. Этот бэкенд также может быть интерпретирован как красивый принтер кода EO и широко используется в качестве такового в других модулях.

- Parser, который содержит парсер (также известный как синтаксический анализатор) исходного кода EO. Он используется для преобразования различных представлений EO (например, обычного текста или кодировки XML) в синтаксическое дерево EO, определенный в модуле Core.

Глава 4

Реализация

В этой главе дается подробное описание реализации анализаторов промежуточного представления EO. Раздел 4.1 описывает общие структуры данных, используемые анализаторами. Разделы 4.2 и 4.3 описывают алгоритмы анализа и их реализации.

4.1 Структуры данных

4.1.1 Синтаксическое дерево EO

Это структура данных, которая используется для моделирования синтаксической структуры EO, которая используется как отправная точка для извлечения более высокоуровневых понятий, таких как класс-объекты, метод-объекты и вызовы методов. Он описывает EO, следуя спецификации синтаксиса из статьи Бугаенко [9], с небольшими отклонениями для учета специфики лежащего в основе уточненного φ-исчисления, описанного в [11].

Для создания синтаксического анализатора от кода EO к синтаксиче-

ского дереву EO мы использовали **cats-parse** [1], монадический синтаксический комбинатор [17], библиотека для Scala.

Синтаксическое дерево EO - это неизменяемая полиморфная структура данных, определяемая *à la carte* [18] (Рис. 4.1). Поскольку дерево неизменяемо, оно может быть изменено только путем построения новой версии, в которой старые части дерева заменяются новыми. Преобразования, которые строят эти новые версии структуры данных, известны как *optics* [19]. Мы использовали *monocle* [2] библиотеку для Scala, чтобы упростить генерацию оптик, которые изменяют синтаксическое дерево EO.

4.1.2 Объектное дерево

Дерево объектов является структурой данных, которая фиксирует отношения между объектами в программе EO. Оно является уточнением синтаксического дерева EO, которое содержит элементы программы EO, относящиеся к последующим шагам анализа: классы-объекты, методы-объекты, пункты расширения и вызовы методов.

Дерево объектов EO также является рекурсивно-определенной полиморфной структурой данных (Рис. 4.2). Параметр типа *A* представляет информацию, которая хранится для каждого объекта в дереве. Эта информация хранится в поле *info* дерева. Поле *nestedObjs* хранит информацию обо всех вложенных классах-объектах. Вложенные объекты - это классы-объекты, которые определяются как атрибуты других объектов класса, подобно вложенным классам в Java. Информация об одном из вложенных объектов может быть доступна по ключу, который имеет тип *Name*. Это имя однозначно иден-

[1] `https://github.com/typelevel/cats-parse`
[2] `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]
```

Рис. 4.1: Определения синтаксического дерева EO (сокращенно)

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

Рис. 4.2: Объектное дерево

тифицирует объект, так как объект не может содержать два атрибута с одинаковым именем.

4.1.3 ObjectInfo

Первый и наиболее общеприменимый тип, который мы используем вместо параметра типа *A* - это *ObjectInfo* (Рис. 4.3). Этот тип также имеет два параметра типа. Первый, *P*, отвечает за хранение информации о декорированном объекте (или просто родителе). Первый обход может собрать только имя декорированного объекта.

Второй параметр типа, *M*, - это тип, используемый для хранения информации о каждом из методов. Информация, полученная во время первого обхода синтаксического дерева EO (Рис. 4.4), может быть обобщена следующим образом:

- *selfArgName* - имя свободного атрибута объекта метода, который используется для захвата вызывающего объекта.
- *body* - узел синтаксического дерева EO, который содержит тело метода.
- *depth* - насколько глубоко вложен объект метода в программу EO. Для объектов верхнего уровня этот атрибут равен 0. Для объектов методов (которые всегда определены в классах-объектах), этот атрибут равен глубине класса-объекта плюс один.

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

Рис. 4.3: ObjectInfo

- *calls* - последовательность вызовов метода в определении метода. Тип *Call* хранит всю необходимую информацию для идентификации и обхода всех вызовов в теле метода:
  - depth - глубина объекта, в котором находится вызов. Это значение равно глубине метода-объекта + относительная глубина метода-локального объекта, содержащего вызов.
  - *methodName* - простое имя метода-объекта, в котором находится вызов.
  - *callSite* - оптика [19], которая извлекает местоположение метода-локального объекта, где находится вызов.
  - *callLocation* - оптика [19], которая извлекает местоположение узла синтаксического дерева EO, который определяет вызов метода.
  - *args* - узлы синтаксического дерева EO, которые соответствуют аргументам вызова, включая аргумент **self**.

```
final case class MethodInfo(
    selfArgName: String ,вызовы
    : Vector[Call],тело
    : EOObj[EOExprOnly],глубина
    : Int ,
)

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

Рис. 4.4: МетодИнфо и вызов

4.1.4 Частичное дерево объектов

ObjectInfo, где *P* - имя родительского объекта, а *M* - *MethodInfo*, можно назвать *частичным деревом объектов*:

```
тип
 PartialObjectTree = ObjectTree[
    ObjectInfo[ParentName, MethodInfo]
]
```

4.1.5 Полное дерево объектов

Так называемое *полное дерево объектов* определяется как следующий псевдоним типа:

```
type CompleteObjectTree =
  ObjectTree[
```

```
    ObjectInfo[
      LinkToParent ,
      MethodInfo
    ]
  ]
```

Единственное, что отличает его от *частичное дерево объектов*, это то, что имя родителя заменяется специальным типом *LinkToParent*. Этот тип также является оптическим [19] и по сути является функцией типа:

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

Берется все дерево объектов программы и возвращается объект, который представляет собой декорированный объект объекта, на который находится ссылка.

4.2 Нахождение непредвиденной взаимной рекурсии

4.2.1 Предлагаемое решение

Решение проблемы заключается в обнаружении циклов в графах вызовов всех объектов. Для каждого класса-объекта в программе сделайте следующее:

1. Определить декорированный класс-объект, все методы, и для каждого метода в классе определить все методы, которые он вызывает. Если вызываемый метод существует в классе-объекте, пометьте его как *решен-*

ный. В противном случае, пометьте его как *частично решенный*. Набор отображений между методами класса и методами, которые каждый из методов вызывает, считается *частичным графом вызовов* объекта.

2. После этого дерево снова обходится, чтобы преобразовать все частично разрешенные вызовы в полностью разрешенные. Для этого нужно вычислить *полный граф вызовов* объекта, который содержит методы из самого объекта, а также методы из декорированного объекта. Это делается путем *расширения* частичного call-графа декорированного объекта с частичным call-графом декорирующих объектов. Здесь и далее мы используем термины **ребенок** и **родитель** для обозначения декорирующего объекта и декорируемого объекта соответственно. Процедура расширения определяется следующим образом:

   (a) если метод присутствует в родительском колл-графе, но отсутствует в дочернем колл-графе, он оставляется как есть.

   (b) если метод присутствует в дочернем колл-графе, но не существует в родительском колл-графе, он добавляется в родительский колл-граф.

   (c) если метод присутствует и в дочернем, и в родительском колл-графе, все вхождения метода в родительском колл-графе заменяются дочерней версией метода.

3. После того, как граф вызовов объекта разрешен, выполните поиск в глубину по первому слову [20], чтобы найти циклы в полном графе вызовов. После того, как все циклы будут найдены, исключите циклы, которые содержат только методы одного и того же объекта.

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

.

Рис. 4.5: *ObjectTreeWithResolvedCallgraphs*

4.2.2 Реализация

Первый шаг алгоритма в 4.2.1 заключается в преобразовании кода EO в *полное дерево объектов* (раздел 4.1.5). Затем объектное дерево преобразуется в вариацию объектного дерева, где все вызовы разрешены. Эта вариация называется *ObjectTreeWithResolvedCallgraphs* (Рис. 4.5). Для каждого объекта хранится только его полностью разрешенный граф вызовов. Последний шаг обходит дерево и собирает все циклы в список специальных объектов типа *CallChain*. Этот объект представляет собой последовательность полностью вычисленных имен методов, которые при вызове никогда не завершаются. Наконец, каждая из таких цепочек вызовов представляется пользователю в виде человекочитаемых строк.

4.3 Нахождение необоснованного предположения в подклассах

4.3.1 Предлагаемое решение

Мы предлагаем следующий подход для обнаружения методов, в которых инлайнинг вызовов может привести к ломающим изменениям в подклассах:

1. Создается *изначальное* представление программы. Это представление представляет собой древовидную структуру данных, которая сохраняет отношения вложенности между объектами. Так, объекты, которые содержат другие объекты, являются корнями соответствующих поддеревьев, в то время как объекты-контейнеры являются поддеревьями или листьями.

2. Мы создаем *ревизию* исходного представления программы, где все вызовы методов инлайнированы. В обеих версиях для каждого класса-объекта, для каждого метода в классе-объекте, определяется набор *свойств*. Эти свойства можно рассматривать как неявный контракт [21] каждого метода. В дополнение к неявным свойствам, учитываются и явные свойства, которые приходят в виде *assert* утверждений в исходном коде. Для того чтобы определить свойства метода, выполняется частичная интерпретация его тела. Интерпретация ограничивается основными числовыми операциями, числовыми и булевыми значениями и вызовами методов. Более подробно правила вывода описаны на рис. 4.6.

3. После того, как все свойства выведены, следующий предикат должен быть истинным как для первоначальной, так и для измененной версии:

$$P_{init} \Rightarrow P_{rev} \tag{4.1}$$

Если это не выполняется для какого-то класса-объекта, это означает, что пересмотр одного из его суперклассов вносит разрывное изменение, которое ослабляет предусловие некоторых его методов.

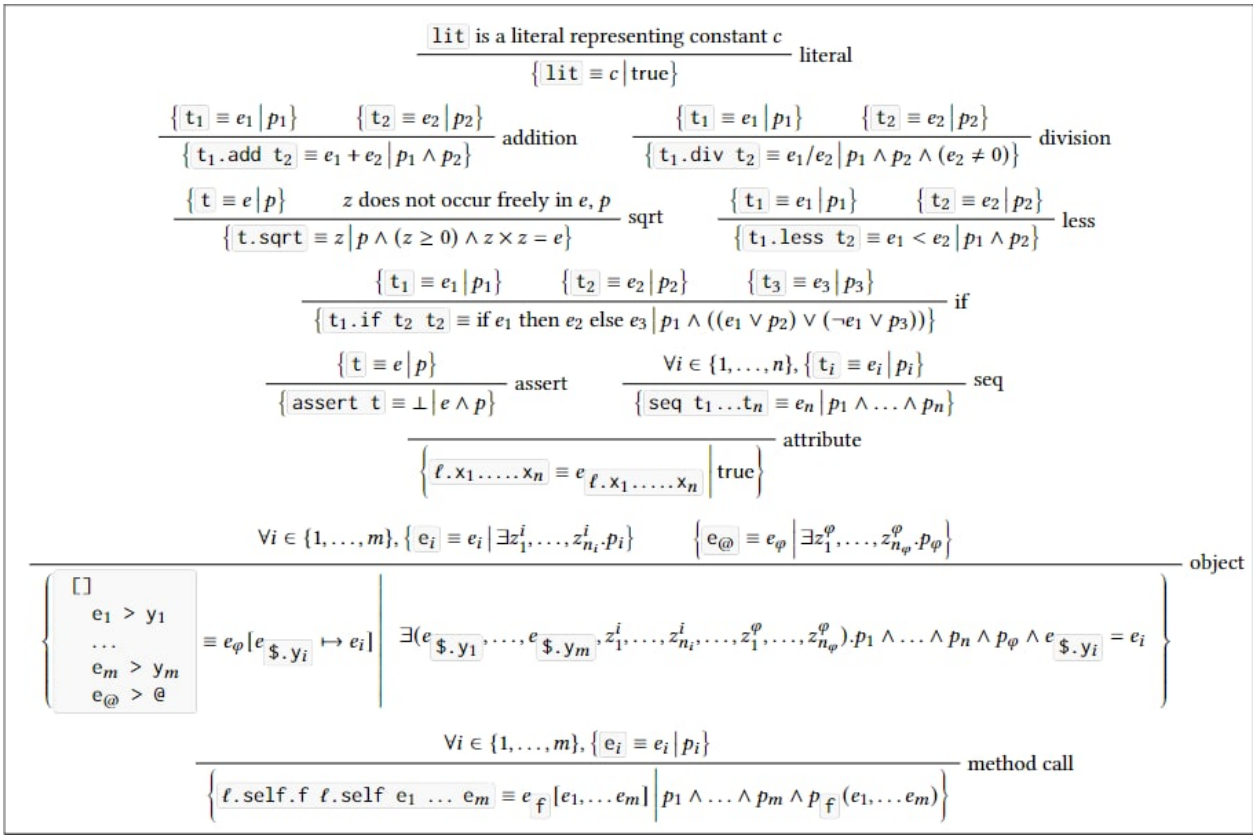

Рис. 4.6: Правила вывода свойств при обнаружении необоснованного предположения в подклассе.

4.3.2 Реализация

Аналогично алгоритму обнаружения взаимной рекурсии, древовидное представление начальной ревизии программы EO осуществляется путем преобразования ее исходного кода в *полное дерево объектов* (раздел 4.1.5). Однако для того, чтобы произвести ревизию исходной программы, нам потребовалось реализовать процедуру инлайнинга для синтаксического дерева EO. Эта процедура может быть представлена следующим образом:

1. Определить все вызовы методов. Это уже сделано во время построения полного дерева объектов.

2. Заменить каждый вызов метода в теле метода значением его φ-атрибута (символ @ в eo).

3. Если метод-объект, который декорируется, содержит атрибуты, отличные от φ, то:

   (a) собирает эти атрибуты в отдельный объект под названием *local_attrs*. Если атрибут с таким именем уже существует в объекте, в который вставляется вызов, разрешите коллизию, добавив суффикс к вновь созданному объекту.

   (b) Добавить этот объект в качестве локального атрибута к методу-объекту, в котором находится вызов.

   (c) Перенаправить ссылки на локальные атрибуты внутри атрибута φ метода, который инлайнится, на их соответствующие атрибуты в объекте *local_attrs*.

4. После того, как пересмотр исходного кода EO получен, исходное дерево объектов и пересмотренное дерево объектов сшиваются вместе в

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

Рис. 4.7: Структура данных для хранения производных свойств.

отдельный экземпляр дерева объектов (Рис. 4.8).

5. Теперь, когда у нас есть вся необходимая информация о первоначальной и пересмотренной версиях, нам нужно вывести их свойства (Рис. 4.6). Эти свойства закодированы в виде программ SMTLIB2 [22]. Мы используем **scala-smtlib** [3] библиотеку для программного построения SMTLIB2-совместимых программ. Эти программы хранятся во вспомогательной структуре данных 4.7. Эта структура имеет следующие поля.

6. Когда эта структура построена как для первоначальной, так и для пересмотренной версии, строится оператор импликации, соответствующий формуле (Рис. 4.1) и передается бэкенду SMT-решателя. Бэкенд, который мы используем в данной диссертации, называется Princess [23].

7. Если решатель находит данное предложение удовлетворяющим, то ошибки нет, и ревизия считается безопасной. В противном случае ревизия считается небезопасной и о ней сообщается пользователю.

[3] https://github.com/regb/scala-smtlib

```
final case class MethodInfoForAnalysis(
  selfArgName: String ,тело
  : EOObj[EOExprOnly] ,глубина
  : Int
)

final case class ObjectInfoForAnalysis[P, M](
  methods: Map[EONamedBnd, M],
  parentInfo: Option[P],
  name: Name,
  indirectMethods: Map[
    Name,
    MethodInfoForAnalysis
  ],
  allMethods: Map[
    Name,
    MethodInfoForAnalysis
  ]
)

type AnalysisInfo = ObjectInfoForAnalysis[
  LinkToParent,
  MethodInfo
]
type InitialAndRevised = ObjectTree[
    (AnalysisInfo, AnalysisInfo)
  ]
```

Рис. 4.8: Дерево объектов, используемое в анализе необоснованных предположений, которое содержит пересмотренную версию объекта вместе с первоначальной версией.

Глава 5

Заключение

В настоящее время парадигма объектно-ориентированного программирования является одним из наиболее доминирующих инструментов в арсенале компаний-разработчиков программного обеспечения. Попытки угнаться за постоянно растущим спросом на инновации привели к неизбежному росту сложности программного обеспечения. Объектно-ориентированные кодовые базы, пожалуй, больше всего страдают от этой сложности, порождая такие явления, как ”унаследованный код”[24]. Существует множество подходов к решению этой проблемы. Один из наиболее часто используемых называется статическим анализом - рассуждениями о коде без его выполнения.

В данной диссертации описывается инновационный подход к статическому анализу объектно-ориентированных программ с использованием φ-calculus - формализации семантики объектно-ориентированных программ, основанной на идее паттерна декоратора [4]. Мы изучили существующие работы по φ-calculus и его реализации, EO [9], описали один из вариантов φ-calculus и применили его для построения статического анализатора, обнаруживающего проблемы семейства ”хрупких базовых классов”[10]. Реализация

была задокументирована, а исходный код опубликован в открытом репозитории Github.

Анализатор был всесторонне протестирован на собственноручно написанных примерах. В настоящее время в конвейере непрерывной интеграции выполняется 89 тестов, некоторые из которых приближаются к 700 строкам кода EO.

5.1 Вклад этой работы

- Парсер для подмножества грамматик EO, описанных в [9].
- Набор полезных структур данных для анализа программ, переведенных в промежуточное представление EO.
- Два алгоритма, использующие вышеуказанные структуры данных для обнаружения двух дефектов семейства ”хрупких базовых классов-непредвиденной взаимной рекурсии и неоправданного допущения в подклассе.

5.2 Будущая работа

Была проделана значительная работа, однако остаются проблемы, которые необходимо решить, прежде чем анализаторы, использующие подход, описанный в данной диссертации, смогут находить дефекты в объектно-ориентированных кодовых базах промышленного масштаба.

- Анализаторы, описанные в данной диссертации, должны быть соединены с транслятором с целевого языка (например, Java) в промежуточное

представление EO. Нам известно о нескольких существующих работах в этой области, однако было проделано мало работы, чтобы заставить их работать вместе с анализаторами.

- Выполнение сложных анализов, таких как выявление необоснованного предположения в подклассе, значительно затруднено из-за отсутствия в EO средства проверки типов. Наличие информации о типах объектов во время анализа сделало бы процесс вывода ограничений, подобный описанному в 4.3.1, более надежным и точным.

- Анализаторы успешно работают на небольших тестовых примерах, однако их тестирование на больших массивах кода EO, например, сгенерированных из существующего исходного кода Java, может выявить критические недостатки текущего дизайна, а также узкие места в производительности, которые трудно обнаружить на небольших тестовых примерах.

- Текущая кодировка, используемая для перевода элементов объектно-ориентированных программ в EO 2.4, хотя и является достаточно общей для представления простых объектно-ориентированных программ, не отражает всей полноты возможностей, присутствующих в объектно-ориентированных языках программирования. Необходимо провести дополнительные исследования, чтобы разработать более полную кодировку.

- Текущий дизайн анализаторов не учитывает сообщения об ошибках, поэтому их текущие возможности ограничены полностью вычисленными именами объектов, в которых произошли ошибки. Необходимо пересмотреть дизайн, чтобы сообщения об ошибках указывали на точные

места ошибок в промежуточном представлении EO и, соответственно, в исходном коде целевого языка.

Список литературы

[1] I. D. Craig, *Object-oriented programming languages: interpretation*. Springer, 2007.

[2] S. L. Ram и др., «Dr. Alan Kay on the Meaning of”Object-Oriented Programming”,» 2003.

[3] R. Mansfield, «Has OOP Failed?,» 2005.

[4] E. Gamma, R. Helm, R. E. Johnson и J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[5] M. Fowler и M. Foemmel, *Continuous integration*, 2006.

[6] C. Lattner и V. Adve, «LLVM: A compilation framework for lifelong program analysis & transformation,» в *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, IEEE, 2004, с. 75—86.

[7] R. Vallee-Rai и L. J. Hendren, «Jimple: Simplifying Java bytecode for analyses and transformations,» 1998.

[8] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam и V. Sundaresan, «Soot: A Java bytecode optimization framework,» в *CASCON First Decade High Impact Papers*, 2010, с. 214—224.

[9] Y. Bugayenko, «EOLANG and phi-calculus,» *CoRR*, т. abs/2111.13384, 2021. arXiv: 2111.13384. url: https://arxiv.org/abs/2111.13384.

[10] L. Mikhajlov и E. Sekerinski, «A study of the fragile base class problem,» в *ECOOP'98 — Object-Oriented Programming*, E. Jul, ред., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, с. 355—382, ISBN: 978-3-540-69064-1.

[11] N. Kudasov и V. Sim, «Formalizing $φ$-calculus: a purely object-oriented calculus of decorated objects,» 2022. DOI: 10.48550/ARXIV.2204.07454. url: https://arxiv.org/abs/2204.07454 (дата обр. 29.05.2022).

[12] L. Räihä, «Delegation: dynamic specialization,» en, в *Proceedings of the conference on TRI-Ada '94 - TRI-Ada '94*, Baltimore, Maryland, United States: ACM Press, 1994, с. 172—179, ISBN: 9780897916660. DOI: 10.1145/197694.197718. url: http://portal.acm.org/citation.cfm?doid=197694.197718 (дата обр. 27.05.2022).

[13] Y. Bugayenko, «Reducing Programs to Objects,» 2021. DOI: 10.48550/ARXIV.2112.11988. url: https://arxiv.org/abs/2112.11988 (дата обр. 07.06.2022).

[14] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner и M. Oivo, «Continuous deployment of software intensive products and services: A systematic mapping study,» *Journal of Systems and Software*, т. 123, с. 263—291, 2017.

[15] W. Ke и K.-H. Chan, «Pattern Matching Based on Object Graphs,» *IEEE Access*, т. 9, с. 159 313—159 325, 2021, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3128575. url: https://ieeexplore.ieee.org/document/9617454/ (дата обр. 31.05.2022).

[16] A. Kennedy и C. V. Russo, «Generalized algebraic data types and object-oriented programming,» en, *ACM SIGPLAN Notices*, т. 40, № 10, с. 21—40, окт. 2005, ISSN: 0362-1340, 1558-1160. DOI: 10.1145/1103845.1094814. url: https://dl.acm.org/doi/10.1145/1103845.1094814 (дата обр. 31.05.2022).

[17] S. Hill, «Combinators for parsing expressions,» en, *Journal of Functional Programming*, т. 6, № 3, с. 445—464, май 1996, ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796800001799. url: https://www.cambridge.org/core/product/identifier/S0956796800001799/type/journal_article (дата обр. 24.05.2022).

[18] W. SWIERSTRA, «Data types à la carte,» *Journal of Functional Programming*, т. 18, № 4, с. 423—436, 2008. DOI: 10.1017/S0956796808006758.

[19] B. Clarke, D. Elkins, J. Gibbons, F. Loregian, B. Milewski, E. Pillmore и M. Román, «Profunctor Optics, a Categorical Update,» 2020. DOI: 10.48550/ARXIV.2001.07488. url: https://arxiv.org/abs/2001.07488 (дата обр. 04.06.2022).

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest и C. Stein, «Depth-first search,» *Introduction to algorithms*, с. 540—549, 2001.

[21] B. Meyer, *Touch of Class*, en. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, ISBN: 9783540921448 9783540921455. DOI: 10.1007/978-3-540-92145-5. url: http://link.springer.com/10.1007/978-3-540-92145-5 (дата обр. 31.05.2022).

[22] C. Barrett, P. Fontaine и C. Tinelli, *The Satisfiability Modulo Theories Library (SMT-LIB)*, www.SMT-LIB.org, 2016.

[23] P. Rümmer, «A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic,» в *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, сер. LNCS, т. 5330, Springer, 2008, с. 274—289, ISBN: 978-3-540-89438-4.

[24] M. Feathers, *Working Effectively with Legacy Code: WORK EFFECT LEG CODE _p1*. Prentice Hall Professional, 2004.