

Санкт-Петербургский государственный университет

Кафедра системного программирования

Мухин Артем Михайлович

Статический анализ кода в IntelliJ Rust

Дипломная работа

Научный руководитель:
доцент Литвинов Ю. В.

Консультант:
программист ООО «ИнтеллиДжей Лабс»
Бескровный В. Н.

Рецензент:
координатор образовательных проектов ООО «ИнтеллиДжей Лабс»
Кладов А. А.

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering Chair

Artem Mukhin

Static code analysis in IntelliJ Rust

Graduation Thesis

Scientific supervisor:
assistant professor Litvinov Y. V.

Consultant:
software developer at IntelliJ Labs Co. Ltd
Beskrovnyy V. N.

Reviewer:
educational project coordinator at IntelliJ Labs Co. Ltd
Kladov A. A.

Saint-Petersburg
2019

Оглавление

Введение	4
1. Постановка задачи	6
2. Анализ программ	7
3. Существующие решения	9
4. Модель владения в языке Rust	11
5. Категоризация памяти	13
6. Построение графа потока управления	14
7. Анализ потока данных	15
8. Анализ перемещений	19
9. Интеграция в IntelliJ Rust	22
10. Апробация и тестирование	25
Заключение	26
Список литературы	28

Введение

Rust — современный язык системного программирования. Он появился в 2010 году и разрабатывается исследователями Mozilla Research.

Небольшой размер среды исполнения (runtime) и ручное управление памятью (без сборщика мусора) ставят Rust в один ряд с языками C и C++. При этом, в отличие от старых языков системного программирования, Rust имеет ряд важных преимуществ.

Во-первых, Rust имеет богатую систему типов, основанную на системе Хиндли-Милнера; Rust сильно типизирован и поддерживает автоматический вывод типов и сопоставление с образцом. В силу этих особенностей Rust достаточно близок к OCaml: оба языка позволяют сочетать императивную и функциональную парадигмы программирования и избегать ошибок, связанных с типами, которые обычно встречаются в программах на C и C++.

Однако, в отличие от компиляторов большинства функциональных языков, таких как OCaml и Haskell, которые полагаются на сборщик мусора и не позволяют работать с памятью вручную, компилятор Rust вынужден гарантировать правильность работы с памятью статически. За это отвечает анализатор заимствований (borrow checker) — статический анализатор кода, встроенный в компилятор. Анализ заимствований основан на анализе потока данных (data-flow) и проверяет соответствие программы некоторым правилам перемещений (move) и заимствований (borrow).

IntelliJ Rust — это плагин для поддержки языка Rust в интегрированных средах разработки (IDE), основанных на платформе IntelliJ: IntelliJ IDEA, CLion, PyCharm и др. Этот плагин разрабатывается в компании JetBrains и имеет открытый исходный код, доступный на GitHub ¹.

В IntelliJ Rust уже реализовано множество возможностей, таких как навигация по коду, разрешение имен, вывод типов, автодополнение кода, интеграция с отладчиками и др. Помимо этого, программистам бы-

¹<https://github.com/intellij-rust/intellij-rust>

ло бы полезно видеть понятные сообщения об ошибках, связанных с перемещениями и заимствованиями, во время написания кода, и иметь возможность автоматически исправить их до сборки всего проекта.

1. Постановка задачи

Целью данной работы является реализация анализатора перемещений для языка Rust и его внедрение в плагин IntelliJ Rust. Для достижения этой цели были поставлены следующие задачи:

- Разработать анализатор, в том числе реализовать:
 - категоризацию памяти;
 - построение графа потока управления;
 - анализ перемещений;
- Внедрить анализатор в плагин IntelliJ Rust;
- Провести апробацию.

2. Анализ программ

Ошибки в программах — серьезная проблема, затрагивающая как программистов, так и пользователей программ и общество в целом. Известно множество случаев, когда ошибки в программном коде приводили к настоящим катастрофам.

Поэтому очень важно проверять программы на наличие в них ошибок до их использования. Для поиска ошибок придумано множество подходов: тестирование, рецензирование, анализ. Данная работа посвящена анализу кода, поэтому рассмотрим подробнее это понятие.

Анализ программ — это процесс автоматического исследования программы с целью проверить некоторые её свойства (например, корректность или отказоустойчивость). Традиционно выделяют два типа анализа кода: *динамический* и *статический*.

В рамках динамического анализа исследуется не исходный код, а сама программа в момент её исполнения. Этот вид анализа требует заданного набора данных, которые будут поданы на вход исследуемой программе. Поэтому эффективность динамического анализа зависит от полноты этих данных, и точность результата анализа, как правило, не гарантируется.

Статический анализ, напротив, использует только исходный код программы (или его промежуточное представление). Помимо поиска ошибок в программе, он может проверять код на соответствие заданному стилю или стандарту оформления и собирать метрики (количество строк кода, сложность потока управления и другие). Статический анализ не нуждается в наборе входных данных и обычно осуществляет полное покрытие исследуемого кода.

Однако в общем случае задача статического анализа является неразрешимой. Это следует из теоремы Райса, которую можно сформулировать для Тьюринг-полных языков так: задача проверки любого нетривиального семантического свойства программы является неразрешимой [2]. Поэтому эффективность статического анализа зависит от анализируемого языка, в том числе от его семантики и системы типов.

На сегодняшний день статический анализ широко используется не только в компиляторах [18] и автономных инструментах анализа [5], но и в интегрированных средах разработки (IDE). Одной из наиболее популярных сейчас IDE является IntelliJ IDEA — среда разработки, предназначенная для разработки на языке Java (а также других языках). В IntelliJ IDEA реализован широкий набор статического анализа: определение мертвого и недостижимого кода, поиск использований неинициализированных переменных и т.д [4]. Помимо поиска ошибок, IDEA в некоторых случаях предлагает автоматические исправления (quick-fixes). На рисунке 1 показан пример статического анализа в IntelliJ IDEA.

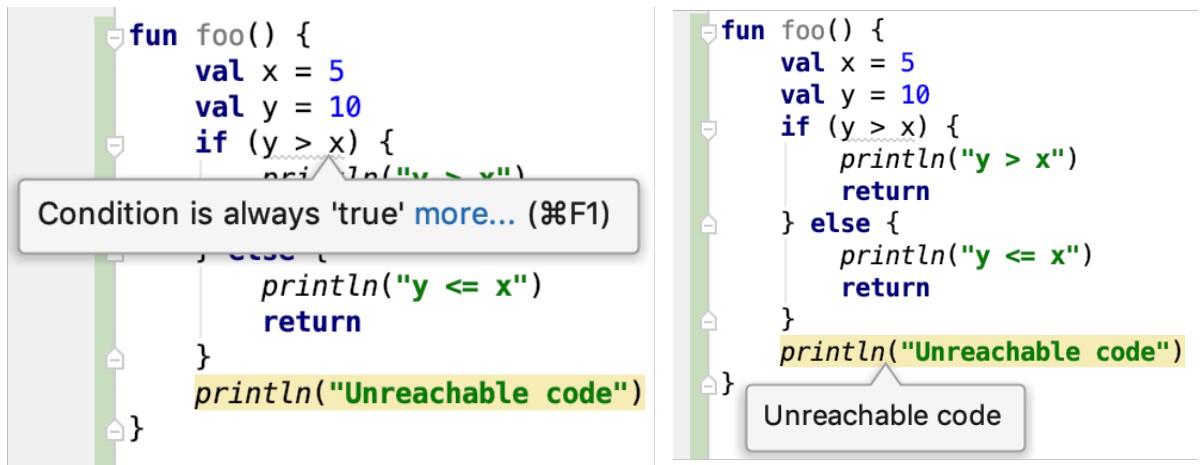


Рис. 1: Статический анализ в IntelliJ IDEA

3. Существующие решения

Помимо IntelliJ Rust существует множество редакторов кода, поддерживающих Rust с помощью плагинов: Visual Studio Code, Vim, Emacs и др. Наиболее популярные плагины для них используют Rust Language Server (RLS) [6] – реализацию Language Server Protocol [7] для языка Rust. RLS позволяет редакторам получать информацию о коде, необходимую для работы различных действий и инспекций в редакторе: перехода к определению, поиска использований символа, показа типа выражения, форматирования кода и т. д. Эту информацию RLS получает от компилятора Rust (`rustc`).

Подход использования компилятора как бэкенда для IDE отлично работает в некоторых случаях. Например, CLion использует для анализа кода `clangd` – реализацию Language Server Protocol на основе Clang. Visual Studio использует компилятор Roslyn [11]. Однако в отличие от `clangd` и Roslyn, компилятор `rustc` не проектировался таким образом, чтобы помогать IDE. Во-первых, компилятор Rust не умеет работать в режиме сервера: при каждом изменении исходного кода приходится перезапускать компиляцию проекта в новом процессе. Во-вторых, `rustc` не предоставляет необходимый для среды разработки API: например, нельзя получить варианты автодополнения кода. В третьих, `rustc` не умеет анализировать синтаксически некорректный код: если пользователь забыл или не успел дописать один символ, то RLS не сможет предоставить результаты семантического анализа кода, а покажет только ошибку парсинга.

Таким образом, RLS не позволяет анализировать код «на лету» (прямо во время ввода символов), а также лишен одной из важнейших возможностей современных IDE – автодополнения кода на основе типов и областей видимости; вместо этого автодополнение RLS основано на регулярных выражениях.

Целью проекта IntelliJ Rust является создание полноценной IDE для Rust, в нем не используется RLS. Вместо этого вся необходимая функциональность реализуется внутри плагина: парсер (который умеет ра-

ботать с неполным кодом), разрешение имен, вывод типов и так далее. Анализ кода инкрементален (его результаты кэшируются), и пользователи видят ошибки сразу же в момент написания кода; им не приходится запускать анализ отдельно и ждать его результатов. Поэтому чтобы сообщать об ошибках перемещений и заимствований, необходимо реализовать эту часть компилятора внутри плагина.

В декабре 2017 года благодаря усилиям Алексея Кладова, в прошлом одного из основных разработчиков IntelliJ Rust, возник проект Rust Analyzer[13], который также называют RLS 2.0. В отличие от RLS, Rust Analyzer не просто запускает rustc и использует его вывод, а самостоятельно реализует основные этапы компиляции, необходимые IDE. Данный проект очень интересен и важен для языка Rust, поскольку на его основе можно будет реализовать более совершенные плагины для поддержки Rust в любых текстовых редакторах и IDE. Однако на момент написания работы Rust Analyzer ещё не готов для полноценного использования в качестве бэкенда IDE.

4. Модель владения в языке Rust

Мы не будем подробно рассматривать синтаксис и семантику языка Rust [14, 15], поскольку анализ перемещений связан прежде всего с системой типов [9]. Ключевой особенностью системы типов Rust является модель *владения* (также называемая семантикой перемещений), которую можно неформально описать тремя правилами:

1. Каждый ресурс имеет владельца (некоторую переменную)
2. В каждый момент времени владелец у ресурса только один
3. Когда переменная-владелец выходит из области видимости [17], ресурс освобождается

Для демонстрации этих правил рассмотрим пример на языке Rust:

```
1 struct S;  
2 fn foo(a: S) {}  
3  
4 let x = S;  
5 let y = x;  
6 let z = x; // Error: use of moved value  
7 foo(x);    // Error: use of moved value
```

Сначала в результате присваивания переменная x становится владельцем ресурса (созданного на стеке экземпляра структуры S). Далее происходит передача владения: теперь владельцем этого ресурса является переменная y . Теперь переменная x уже не владеет ресурсом, поэтому передача владения от x к z приводит к ошибке компиляции. Такую же ошибку вызовет передача владения в функцию foo (то есть к параметру a).

Переменная считается перемещенной сразу после её объявления; при этом сразу после инициализации переменная перестает быть перемещенной:

```
1 struct S;  
2 let x: S; // x is moved  
3 x = S;    // x is not moved
```

Таким образом, если считать, что после перемещения переменная становится неинициализированной, то понятие *перемещенной* переменной полностью совпадает с понятием *неинициализированной* переменной.

Идея модели владения исходит из понятия аффинной системы типов. Линейная система типов [20] гарантирует, что каждый ресурс будет использован ровно один раз. Аффинная система типов — ослабление линейной, позволяющее также не использовать ресурс ни разу.

В последние годы активно развиваются проекты по формализации системы типов языка Rust, в том числе с доказательствами корректности модели владения [10, 16].

5. Категоризация памяти

Выражения в программе на Rust всегда связаны с некоторой памятью, которая используется при их вычислении. Для анализа перемещений необходимо разделить все выражения на несколько категорий в соответствии со свойствами этой памяти. В процессе категоризации для каждого выражения вычисляется *категория* и *категория изменяемости*².

Категории можно определить индуктивно; составные категории являются композицией базовых. Например, выражение `(*x).a` имеет составную категорию «Обращение к полю разыменованного указателя».

Базовые категории:

- Временное значение (rvalue);
- Адрес локальной переменной;
- Разыменованный указатель;
- Обращение к содержимому (поле, элемент).

Категорий изменяемости всего три:

- Неизменяемое;
- Изменяемое по определению;
- Изменяемое как содержимое изменяемого.

Процесс категоризации реализован в виде рекурсивного обхода синтаксического дерева, который запускается после вывода типов. В результате этого процесса для каждого выражения известен не только его тип, но и его категория и категория изменяемости. Результат категоризации кэшируется для каждого выражения, поэтому при изменении исходного кода требуется пересчитать только категории измененных выражений.

²<https://doc.rust-lang.org/book/mutability.html>

6. Построение графа потока управления

Граф потока управления [1] — это ориентированный граф, вершинами которого являются операторы и выражения программы, а ребра показывают возможные пути её исполнения. При построении графа потока управления операторы, между которыми нет ветвлений и инструкций перехода, часто объединяют в *базовые блоки*. Однако мы не используем базовые блоки, т. к. анализ перемещений локален на уровне выражений: даже внутри одного оператора могут возникнуть несколько перемещений. Поэтому в нашем случае вершинами графа потока управления могут быть любые выражения и операторы языка, а также специальные вершины: *Entry* (входная вершина), *Exit* (выходная вершина), *Dummy* (фиктивная вершина; используется при обходе конструкций со сложным ветвлением), *Unreachable* (недостижимая вершина).

Построения графа потока управления реализовано в виде рекурсивного обхода синтаксического дерева с помощью шаблона «Посетитель». При обработке некоторого составного выражения сначала создаются вершины для его подвыражений в порядке исполнения, а затем создается вершина для всего выражения.

Чтобы проверить построение графа потока управления, было создано действие в IDE, показывающие граф для выбранной функции с помощью Graphviz³. Пример построенного графа изображен на рисунке 2.

³<https://www.graphviz.org/>

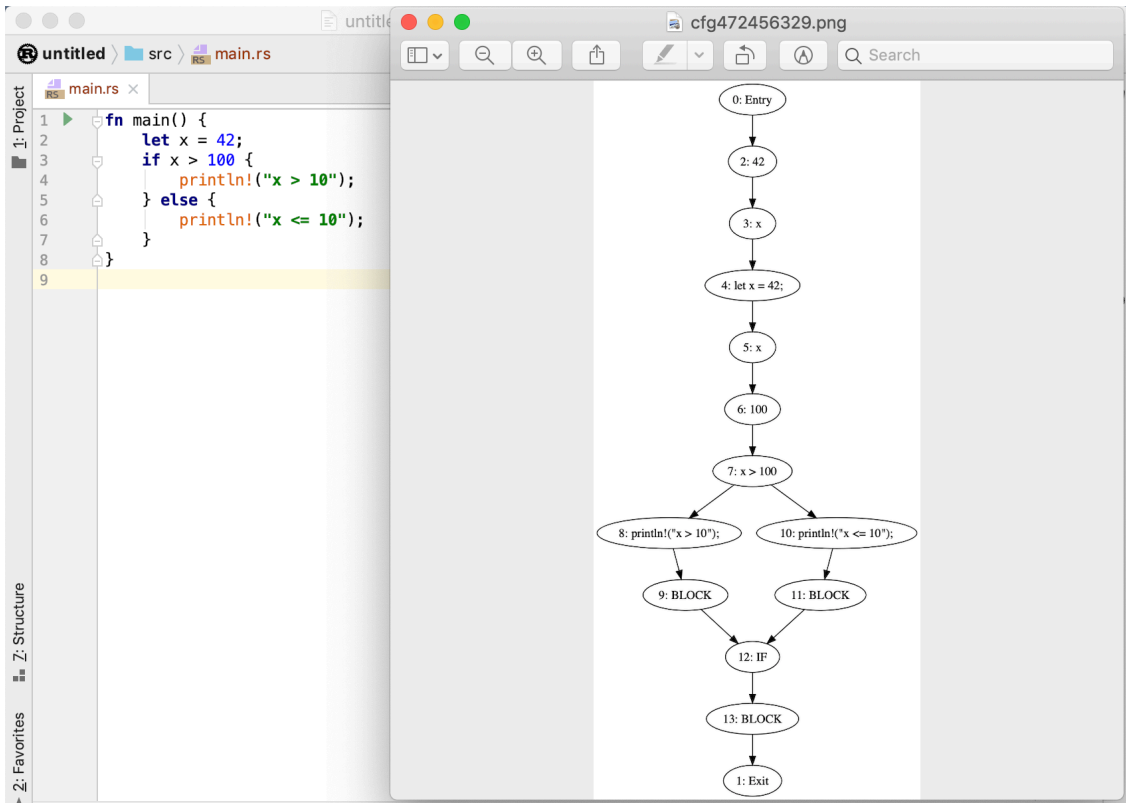


Рис. 2: Пример графа потока управления

7. Анализ потока данных

Анализ перемещений в Rust основывается на анализе потока данных [19, 18]. Такой анализ учитывает не только порядок исполнения программы, но и информацию о том, какие значения вычислены (или определены) в данной точке исполнения. Эту информацию можно представить в виде *решетки*.

Решетки

Решетка[3] — это алгебраическая структура с двумя идемпотентными, коммутативными и ассоциативными операциями \vee и \wedge , удовлетворяющими свойству поглощения:

$$a \vee (a \wedge b) = a$$

$$a \wedge (a \vee b) = a$$

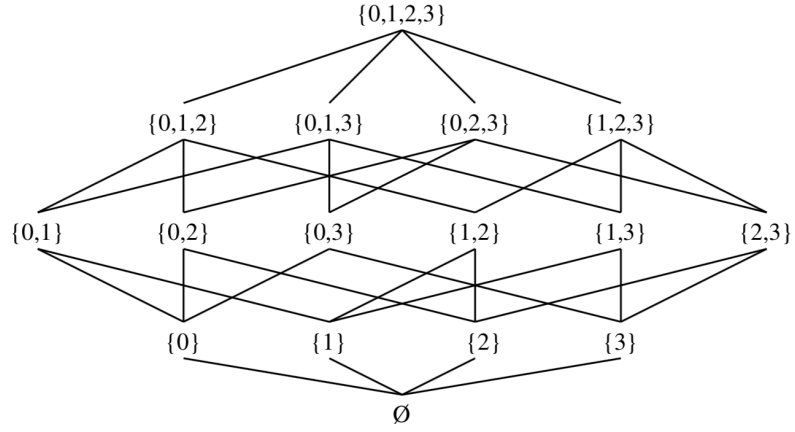


Рис. 3: Булеан множества $\{0, 1, 2, 3\}$ как решетка

На решетке можно установить отношение частичного порядка: $a \sqsubseteq b$, если $a \wedge b = a$. У решетки есть единственный наименьший элемент \perp и единственный наибольший элемент \top . *Высота* решетки – это максимальная длина пути от \perp до \top (в случае решетки подмножеств S это $|S|$).

В дальнейшем мы будем работать с решеткой подмножеств L некоторого конечного множества S (пример такой решетки изображен на рисунке 3). Заметим, что её высота конечна и равна $|S|$.

Функция $f : L \rightarrow L$ называется *монотонной*, если

$$\forall a, b \in S : a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$$

Теорема о неподвижной точке. В решетке конечной высоты L каждая монотонная функция f имеет единственную наименьшую неподвижную точку:

$$\text{fix}(f) = \bigcup_{i \geq 0} f^i(\perp), \quad f(\text{fix}(f)) = \text{fix}(f)$$

Анализ потока данных оперирует с графом потока управления и решеткой конечной высоты, которая используется для представления информации о каждой точке исполнения. Для каждой вершины графа потока управления вводятся ограничения, которые связывают информацию в текущей вершине с информацией в других вершинах. Если

ограничения представляют собой уравнения с монотонными функциями в правых частях, то решение (т. е. результат анализа) существует и может быть найдено по теореме о неподвижной точке.

В ходе работы был реализован модуль прямого анализа потока данных [19]. В качестве реализации множества используются битовые массивы: каждой переменной назначается индекс, по которому в массиве находится либо 1 (переменная принадлежит этому множеству), либо 0 (переменная не принадлежит этому множеству).

Анализ потока данных оперирует с двумя массивами: GEN и KILL. Элементы массива — слова, их размер равен высоте решетки, округленной вверх до числа, кратного 8 (чтобы оперировать с байтами). GEN[i] — i -тое слово массива, соответствует i -ой вершине графа потока управления; GEN[i][k] — k -ый бит i -того слова, соответствует k -тому элементу решетки. Таким образом, GEN хранит отдельное множество элементов решетки для каждой вершины графа. Аналогичное верно для массива KILL.

В массив GEN записывается информация, которая возникла в некоторый момент исполнения, а в массив KILL — информация, которая исчезла в некоторый момент исполнения. Массивы заполняются перед началом анализа потока данных. Сам процесс анализа представляет собой *распространение данных* во время обхода графа потока управления в порядке reverse post-order. Распространение — это построение множеств IN и OUT по следующим формулам:

$$\text{IN}(e) = \bigcup_{p \in \text{preds}(e)} \text{OUT}(p)$$

$$\text{OUT}(e) = \text{GEN}(e) \cup (\text{IN}(e) \setminus \text{KILL}(e)) = f_e(\text{IN}(e))$$

где:

e — текущая вершина графа потока управления

IN(e) — множество элементов решетки в точке исполнения перед e

OUT(e) — множество элементов решетки в точке исполнения после e

Заметим, что GEN и KILL фиксированы, и вычисление $OUT(e)$ монотонно по $IN(e)$. Следовательно, процесс распространения конечен и приводит к единственному наименьшему решению.

8. Анализ перемещений

Основная цель анализа перемещений состоит в том, чтобы проверить, что вся используемая память инициализирована. Для отыскания использований неинициализированной памяти отслеживаются все точки в программе (вершины в графе потока управления), создающие перемещение. Каждой вершине графа соответствует слово в массивах GEN и KILL, биты которого соответствуют каждому перемещению. Присваивания обнуляют биты, созданные перемещением. Когда пути в потоке управления соединяются, то биты объединяются.

Часто при анализе потока данных биты соответствуют переменным [19]. Однако в нашем случае биты соответствуют каждому *перемещению*, а не каждому перемещаемому пути. Этот факт отрицательно влияет на производительность анализа, так как в каждой точке исполнения анализатору приходится искать и проверять все перемещения, связанные с текущей переменной. С другой стороны, такой способ представления удобнее для нас, поскольку при возникновении ошибки IDE может показать конкретное место, где было произведено перемещение. По этой же причине в рамках анализа было бы неудобно работать с более эффективными представлениями кода (например, с формой SSA [18]).

Анализ перемещений состоит из двух этапов: сбор перемещений и проверка перемещений. Каждый из этапов представляет собой рекурсивный обход синтаксического дерева, параметризованный некоторым делегатом. В процессе обхода синтаксического дерева всякое выражение, «потребляющее» (то есть вызывающее копирование или перемещение) ресурс или выполняющее присваивание, передается делегату. В зависимости от этапа анализа делегат, получая некоторое выражение, производит различные действия.

Сбор перемещений

- Проверяет наличие перемещения

Значения в Rust перемещаются по умолчанию кроме некоторых

особенных типов (ссылки, указатели, функции) и типов, реализующих класс типов `Copy`.

- Проверяет возможность перемещения

Некоторые категории памяти перемещать запрещено: элементы массива, статические переменные, разыменованные указатели. При возникновении таких попыток перемещения анализатор сообщает пользователю об ошибке

- Записывает информацию о перемещении

Для каждого перемещения хранится *путь* и выражение, породившее это перемещение (оно используется для сообщения об ошибках). Путь определяется исходя из категории выражения: например, это может быть локальная переменная или поле структуры.

Непосредственно после сбора перемещений происходит построение множеств `GEN` и `KILL`. Рассмотрим процесс их построения на примере простой программы. Напротив каждого оператора показаны соответствующие им значения множеств `GEN` и `KILL` (т. е. множества k , таких что $\text{GEN}[i][k] = 1$ или $\text{KILL}[i][k] = 1$, где i — номер вершины-оператора).

`GEN[i][0]` — биты перемещения $y = x$

`GEN[i][1]` — биты перемещения x ;

```
1 fn foo() {
2     let x = S; // GEN = {}, KILL = {0, 1}
3     let y = x; // GEN = {0}, KILL = {}
4     x;        // GEN = {1}, KILL = {}
5     // GEN = {}, KILL = {0, 1}
6 }
```

В данной программе создается переменная x . Её инициализация уничтожает оба перемещения. Далее значение x перемещается в y , поэтому возникает первое перемещение. Затем возникает второе перемещение x . В конце обе переменные выходят из области видимости и происходит уничтожение соответствующих перемещений.

После этого происходит обход графа и распространение данных. На выходе получаем множества IN и OUT. Ниже показаны значения множества IN для операторов исходной программы:

```
1 fn foo() {
2     let x = S; // IN = {}
3     let y = x; // IN = {}
4     x;         // IN = {0}
5     // IN = {0, 1}
6 }
```

Таким образом, к моменту исполнения последнего оператора в блоке уже произошло первое перемещение, а к концу всего блока — оба перемещения.

После этого наступает этап проверки перемещений. Основная идея этого этапа состоит в том, чтобы при обработке очередного перемещения (а также использования или присваивания) проверять, не перемещена ли соответствующая память до этого. Для этого используются уже построенные множества IN и OUT.

Проверка перемещений

- Проверяет, что память инициализирована

Каждое использование какого-либо значения должно относиться только к инициализированной памяти. Поэтому всякое использование неинициализированного ресурса (в т. ч. после его перемещения или объявления без инициализации) порождает ошибку компиляции.

- Проверяет, что присваивание корректно

Присваивания с чтением (например, +=) должны иметь слева пути, значения которых инициализированы. Поле, в которое происходит присваивание (например, x.a = 1), должно принадлежать инициализированной структуре.

9. Интеграция в IntelliJ Rust

Плаги́н IntelliJ Rust написан на языке Kotlin и использует IntelliJ Platform⁴ как фундамент для взаимодействия с кодом программы и всеми компонентами IDE. Анализатор перемещений тоже написан на Kotlin и не использует никаких дополнительных библиотек.

Анализ перемещений запускается отдельно для каждой функции из исходного кода. Результаты анализа кэшируются, поэтому при изменении тела одной функции анализ будет перезапущен только для неё.

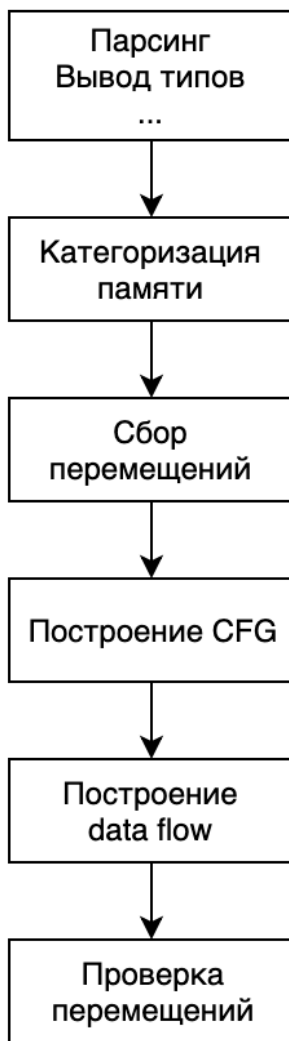


Рис. 4: Схема работы анализатора

⁴<https://www.jetbrains.com/opensource/idea/>

Инспекции и исправления

Ошибки, найденные анализатором, используются в нескольких инспекциях: «Use of moved value», «Use of possibly uninitialized variable», «Cannot move».

```
1
2 ↓ struct S;
3
4 fn f(s: S) {}
5
6 ▶ fn main() {
7     let s = S;
8     f(s);
9     print(s);
10
11
```

Use of moved value [more...](#) (⚠F1)

Рис. 5: Инспекция «Use of moved value»

```
1 fn foo(cond: bool) {
2     let x: i32;
3     if cond {
4         x = 42;
5     }
6     let y = 2 * x + 1;
7
```

Use of possibly uninitialized variable [more...](#) (⚠F1)

Рис. 6: Инспекция «Use of possibly uninitialized variable»

```
struct S;
fn main() {
    let x = [S, S, S];
    let y = x[0];
}
```

Cannot move [more...](#) (⚠F1)

Рис. 7: Инспекция «Cannot move»

При появлении ошибки «Use of moved value» пользователю предлагается автоматически реализовать класс типов `Copy` для соответствующей структуры (см. рисунок 8). При ошибке «Use of possibly uninitialized variable» предлагается инициализировать переменную значением по умолчанию (см. рисунок 9).

```
1 1↓ struct S;  
2  
3 ► fn main() {  
4     let s1 = S;  
5     let s2 = s1;  
6     ! let s3 = s1;  
7 }  
8  
1  #[derive(Clone, Copy)]  
2 1↓ struct S;  
3  
4  ► fn main() {  
5     let s1 = S;  
6     let s2 = s1;  
7     ! let s3 = s1;  
8 }  
9
```

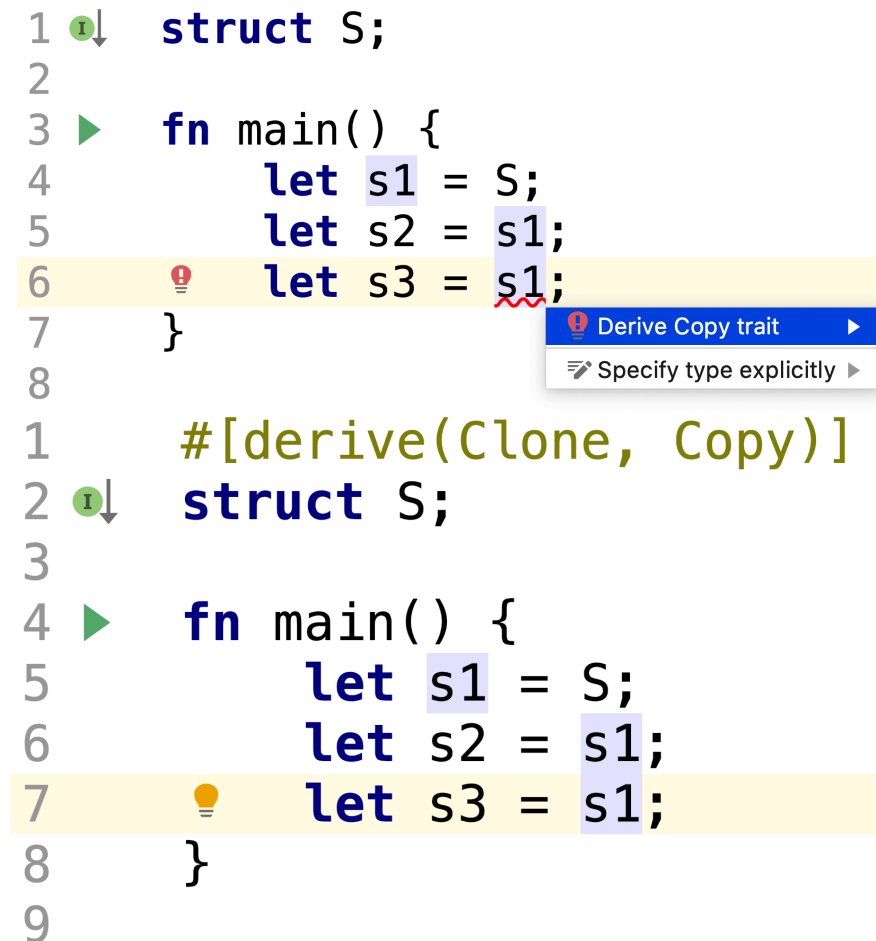
The image shows two snippets of Rust code. The top snippet shows a function `main` with three `let` bindings: `s1 = S`, `s2 = s1`, and `s3 = s1`. The third line has a red error icon and a red squiggly line under `s1`. A tooltip menu is open, showing two options: "Derive Copy trait" (highlighted in blue) and "Specify type explicitly". The bottom snippet shows the same code but with `#[derive(Clone, Copy)]` added above the `struct S` definition. The error icon is now a yellow lightbulb, indicating a suggestion.

Рис. 8: Исправление «Use of moved value»


```
1 ▶ fn main() {
2     let x: i32;
3     let y = x + 42;
4 }
5
```

```
1 ▶ fn main() {
2     let x: i32 = 0;
3     let y = x + 42;
4 }
5
```

Рис. 9: Исправление «Use of possibly uninitialized variable»

10. Апробация и тестирование

Для того, чтобы убедиться в работоспособности, были написаны компонентные тесты для категоризации памяти, построения графа потока управления, трех инспекций и исправлений.

Кроме того, для проверки анализ запускался на реальных проектах, написанных на языке Rust (в том числе Cargo⁵ и Tokio⁶). Было обнаружено и исправлено несколько ложно-положительных срабатываний.

После прохождения тестирования анализатор и вся связанная с ним функциональность была добавлена в стабильную версию плагина IntelliJ Rust.

⁵<https://github.com/rust-lang/cargo>

⁶<https://github.com/tokio-rs/tokio>

Заключение

В ходе данной работы были получены следующие результаты:

1. Реализованы отдельные модули для:
 - категоризации памяти;
 - построения графа потока управления;
 - анализа потока данных;
2. На основе этих модулей реализован анализ перемещений.
3. Вся новая функциональность протестирована и добавлена в стабильную версию плагина IntelliJ Rust.

Дальнейшая работа

Основное направление дальнейшей работы — это реализация полноценного анализатора заимствований. Анализатор перемещений несложно расширить до анализатора заимствований. Основная проблема заключается в том, что в декабре 2018 года вышло глобальное обновление языка Rust [12], включающее в себя новый анализатор заимствований, который поддерживает нелексические времена жизни [8]. Благодаря этому анализ заимствований стал менее строгим и более практичным: теперь некоторые «ошибки заимствований», которые находит старый анализатор, становятся ложно-положительными срабатываниями. Полностью реализовать новый анализатор — отдельная и сложная задача. Возможно, мы найдем способ использовать старый анализатор заимствований и избежать ложных срабатываний.

Несмотря на серьезные изменения в языке, данная работа не теряет актуальности. Во-первых, новый стандарт языка не изменяет семантику перемещений, поэтому анализатор перемещений продолжает работать и соответствовать правилам языка. Во-вторых, помимо анализатора перемещений, в ходе данной работы в плагине IntelliJ Rust был

реализован фундамент для статического анализа Rust-кода: категоризация памяти, построение графа потока управления и анализ потока данных. Эта функциональность позволит в будущем написать другие виды анализа, например, подсветку неиспользованных переменных.

Список литературы

- [1] Allen Frances E. Control Flow Analysis // Proceedings of a Symposium on Compiler Optimization. — New York, NY, USA : ACM, 1970. — P. 1–19. — Access mode: <http://doi.acm.org/10.1145/800028.808479>.
- [2] Asperti Andrea. The Intensional Content of Rice's Theorem // SIGPLAN Not. — 2008. — Jan. — Vol. 43, no. 1. — P. 113–119. — Access mode: <http://doi.acm.org/10.1145/1328897.1328455>.
- [3] Garg Vijay K. Introduction to Lattice Theory with Computer Science Applications. — John Wiley and Sons, Inc, 2015. — Jun. — Access mode: <http://dx.doi.org/10.1002/9781119069706>.
- [4] Jemerov Dmitry. Implementing Refactorings in IntelliJ IDEA // Proceedings of the 2Nd Workshop on Refactoring Tools. — WRT '08. — New York, NY, USA : ACM, 2008. — P. 13:1–13:2. — Access mode: <http://doi.acm.org/10.1145/1636642.1636655>.
- [5] Johnson S. C. Lint, a C Program Checker // COMP. SCI. TECH. REP. — 1978. — P. 78–1273.
- [6] Language Server Protocol. — <https://github.com/rust-lang/rls/>. — Accessed: 12.01.2019.
- [7] Language Server Protocol. — <https://microsoft.github.io/language-server-protocol/>. — Accessed: 12.01.2019.
- [8] Non-lexical lifetimes: introduction. — <http://smallcultfollowing.com/babysteps/blog/2016/04/27/non-lexical-lifetimes-introduction/>. — Accessed: 12.01.2019.
- [9] Pierce Benjamin C. Types and Programming Languages. — 1st edition. — The MIT Press, 2002. — ISBN: 0262162091, 9780262162098.
- [10] Reed Eric W. Patina : A Formalization of the Rust Programming Language. — 2015.

- [11] Roslyn Overview. — <https://github.com/dotnet/roslyn/wiki/Roslyn-Overview>. — Accessed: 12.01.2019.
- [12] Rust 2018. — <https://blog.rust-lang.org/2018/12/06/Rust-1.31-and-rust-2018.html>. — Accessed: 12.01.2019.
- [13] Rust Analyzer. — <https://github.com/rust-analyzer/rust-analyzer>. — Accessed: 12.01.2019.
- [14] Rust book. — <https://doc.rust-lang.org/book>. — Accessed: 12.01.2019.
- [15] Rust reference. — <https://doc.rust-lang.org/reference>. — Accessed: 12.01.2019.
- [16] RustBelt: securing the foundations of the Rust programming language / Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, Derek Dreyer // PACMPL. — 2018. — Vol. 2, no. POPL. — P. 66:1–66:34. — Access mode: <https://doi.org/10.1145/3158154>.
- [17] Scott Michael L. Programming Language Pragmatics. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2005. — ISBN: 0126339511.
- [18] Torczon Linda, Cooper Keith. Engineering A Compiler. — 2nd edition. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2011. — ISBN: 012088478X.
- [19] Uday Khedker Amitabha Sanyal, Sathe Bageshri. Data Flow Analysis: Theory and Practice. — CRC Press, 2009. — ISBN: 0849328802.
- [20] Wadler Philip. Linear Types Can Change the World! — 1993. — 10.