Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Кириллов Илья Олегович

Транслятор из Java в Kotlin

Бакалаврская работа

Научный руководитель:
к.т.н., доцент Литвинов Ю.В.

Консультант:
Подхалюзин А.В.

Рецензент:
к.т.н. Глухих М. И.

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering

Ilya Kirillov

Java to Kotlin Converter

Bachelor's Thesis

Scientific supervisor:
Yurii Litvinov

Adviser:
Alexander Podkhalyuzin

Reviewer:
Mikhail Glukhikh

Saint-Petersburg
2019

Оглавление

Введение

Java — объектно-ориентированный статически типизированный язык общего назначения, транслирующийся в байт-код для виртуальной Java-машины (JVM).

Kotlin [8] — это мультипарадигменный статически типизированный язык общего назначения, целевая платформа которого, так же как и у Java, — это JVM. Kotlin считается более типобезопасным, чем Java (в основном из-за nullable типов), а также более лаконичным. В связи с этим многие компании переходят сейчас с языка Java на язык Kotlin. Среди них компания Google, которая в 2017 году официально объявила [9] поддержку языка Kotlin в своей среде разработки под операционную систему Android — Android Studio, а в 2019 году сделала Kotlin основным языком разработки для Android.

Чтобы сделать переход с Java на Kotlin безболезненнее, в среде разработки в IntelliJ IDEA [6] существует инструмент, позволяющий транслировать код на языке Java в код на Kotlin, называемый Java to Kotlin Converter [2]. Но часто он генерирует некорректный код (примеры приведены в главе 2.3). Одной из основных проблем старой версии транслятора является неправильный вывод nullability типов (а поддержка nullability в Kotlin является одним из основных его преимуществ [10]). Неправильная работа старого транслятора связана с тем, что его архитектура не позволяет писать сложные преобразования над кодом, которые необходимы для корректной работы. Поэтому было принято решение написать новую версию транслятора, который генерировал бы корректный Java код, в отличие от оригинального транслятора.

1. Постановка задачи

Целью данной работы является создание транслятора из Java в Kotlin для среды IntelliJ IDEA, который был бы лишён недостатков старого транслятора. Для чего были поставлены следующие задачи:

- Проанализировать недостатки старой версии транслятора;
- Реализовать транслятор из языка Java в язык Kotlin;
- Провести сравнение нового транслятора со старым.

2. Обзор

2.1. Транслятор

Транслятор [1]— это программа, которая берёт исходный код на одном языке программирования (входном) и превращает в исходный код на другом языке программирования (выходном). Процесс трансляции состоит из набора фаз. Фазы обычно выполняются последовательно, и каждая следующая фаза зависит от результата предыдущей. Типичный список фаз представлен ниже.

- Лексический анализ: исходный код входной программы разбивается на список токенов, где токен — это синтаксический элемент исходного кода программы (например, идентификатор объявления или строковый литерал).

- Синтаксический анализ: полученный на предыдущей фазе список токенов преобразуется в Абстрактное Синтаксическое Дерево (англ. Abstract Syntax Tree — AST) — промежуточное представление программы в виде дерева, в котором каждый узел представляет из себя операцию, а потомки этого узла — аргументы данной операции. Для описания структуры языка используются грамматики (способ описания формального языка). Самый часто используемый тип грамматик для описания синтаксиса языков программирования— это контекстно-свободные грамматики [7].

- Вывод и проверка типов: в данном преобразовании для каждого элемента в дереве, который может иметь тип (то есть для выражений), происходит проверка того, что операции применяются к операндам с корректными типами. Например, выражение `42 + true` в большинстве языков программирования будет некорректно, ибо в них нет такой версии оператора "плюс", что работала бы одновременно с числами и логическими значениями. В таком случае транслятор обычно выдаёт ошибку о несоответствии типов и останавливает процесс трансляции. Также многие языки про-

граммирования поддерживают вывод типов [3] — автоматическое определение типа выражений в зависимости от их контекста. В таком случае также происходит и вывод типов.

- Собственно, преобразование AST так, чтобы по нему можно было в дальнейшем сгенерировать код на выходном языке программирования. Способы преобразования сильно зависят от реализации AST, которое может быть как изменяемым, так и неизменяемым. В первом случае при трансформации дерева модифицируется исходное дерево, во втором — на каждом преобразовании создаётся новое. Для преобразования AST обычно обходится рекурсивно либо с помощью паттерна Visitor, либо с помощью сопоставления с образцом (чаще всего второй способ используется в функциональных языках) и в процессе обхода получается новое дерево, которое затем используется в следующих фазах.

- Различные оптимизации кода.

- Генерация кода: преобразование AST, полученного на предыдущем этапе, в код на выходном языке программирования.

2.2. IntelliJ IDEA

Одной из наиболее популярных сред разработки является IntelliJ IDEA [6]. Это интегрированная среда разработки, поддерживающая большое количество языков программирования, среди которых Java и Kotlin. IntelliJ IDEA использует структуру, называемую PSI (Program Structure Interface) [12], которая отвечает за синтаксический разбор и создание синтаксической и семантической моделей кода и от которой зависит большое число функциональности платформы IntelliJ.

2.3. Старый транслятор

В IntelliJ IDEA есть инструмент для преобразования кода на языке Java в код на языке Kotlin [2]. Рассмотрим его принцип работы. Транс-

лятор представляет из себя набор функций, каждая из которых отвечает за конвертацию элементов определённого типа. Функции сгруппированы по классам, например, один класс содержит функции для обработки элементов верхнего уровня, другой — для обработки выражений и т.д. Каждая такая функция берёт на вход Java PSI элемент и возвращает соответствующий ей узел AST. Процесс трансляции Java дерева начинается с его корня (например, с файла) и затем для каждого ребёнка транслируемого PSI элемента вызывается соответствующая функция, умеющая его транслировать. Данный процесс происходит рекурсивно. Например, функция трансляции Java класса вызывает функции трансляции членов данного класса (полей, методов), а те, соответственно, вызывают методы трансляции для своих детей и так далее. В результате данного однопроходного процесса к исходному коду на языке Java применяются все предполагаемые преобразования и рекурсивно строится AST, которое потом печатается, а затем над полученным кодом происходит постобработка. Рассмотрим недостатки данного подхода:

- отсутствует возможность многоразового прохода над AST, что влечёт за собой сложность написания большого числа преобразований над кодом;
- невозможно разделить преобразования кода на отдельные компоненты, которые бы работали независимо друг от друга, что затрудняет написание новых преобразований и переиспользование старых.

Ввиду этого, помимо сложности добавления новых типов преобразований и исправления старых, старый транслятор часто генерирует некорректный код с большим количеством ошибок компиляции.

Рассмотрим на примере кода, представленного на рис 1. Здесь написана функция, возвращающая массив фиксированной длинны с элементами типа `String`, заполненный значениями `null`. Из-за nullable типов (подробнее в разделе 2.4) языка Kotlin в сконвертированном коде, тип функции `array` должен быть `Array<String?>`, а не `Array<String>` —

```
public String[] array() { return new String[42]; }
```

Исходный код на язке Java

```
fun array(): Array<String> { return arrayOfNulls(42) }
```

Сконвертированный старым транслятором код

Рис. 1: Пример некорректно конвертируемого кода старым транслятором

как сгенерировал бы старый транслятор. Это связано с тем, что невозможно произвести вывод nullability за один шаг во время обработки Java кода, как это делается в старой версии транслятора.

2.4. Nullability в языках Java и Kotlin

Одним из самых часто встречающихся подводных камней во многих языках программирования [10] (включая Java) является попытка произвести доступ к члену класса, который на самом деле является `null` значением (в языке Java мы не можем явно указать, может ли тип принимать значение `null` или нет). Что приводит к выбросу исключения, которое называется в Java (и Kotlin) `NullPointerException`. Благодаря nullable типам языка Kotlin `NullPointerException` в Kotlin возможен только при явном вызове оператора проверки на `null`, который превращает значение nullable типа в значение not-null типа и кидает `NullPointerException`, если значение `null`.

Для Null безопасности в языке Kotlin введены специальные nullable типы. Тип, который помечен как nullable (для этого после имени типа добавляется знак вопроса `?`, например, `String?`), может содержать значение `null`. Типы, не помеченные таким образом, значение `null` содержать не могут.

Проблема с nullability является одной из основных проблем транслятора из Java в Kotlin. Старая версия транслятора не всегда выводила nullability для типов правильно. Для работы с nullability в языке Java существуют следующие инструменты:

- Вывод nullability для Java кода, встроенный в среду IntelliJ IDEA: используется в старой версии транслятора и не во всех случаях может вывести nullability правильно (пример на рис. 1);

- Nullability аннотации [11], которые позволяют явно указать может ли объявление принимать `null`, или же оно всегда `not null`. Данный подход используется только в некоторых проектах, написанных на языке Java;

- Внешние аннотации [4] в среде IntelliJ IDEA позволяют добавить nullability аннотации на уже существующие Java объявления (например, для сторонней библиотеки) и хранить их список во внешнем файле;

- Утилита Nullability Inference Tool [5]: позволяет в некоторых случаях произвести вывод nullability для полей Java классов, основываясь на Java байт-коде.

3. Реализация

Транслятор из Java в Kotlin реализован в среде разработки IntelliJ IDEA в качестве замены старой версии транслятора. Транслятор написан на языке Kotlin. Код из старой версии транслятора практически не переиспользуется, так как старый транслятор работал с Java кодом на уровне PSI дерева, а новый сначала строит по PSI дереву AST, а затем уже анализирует полученное дерево и выполняет над ним преобразования.

3.1. Структура AST

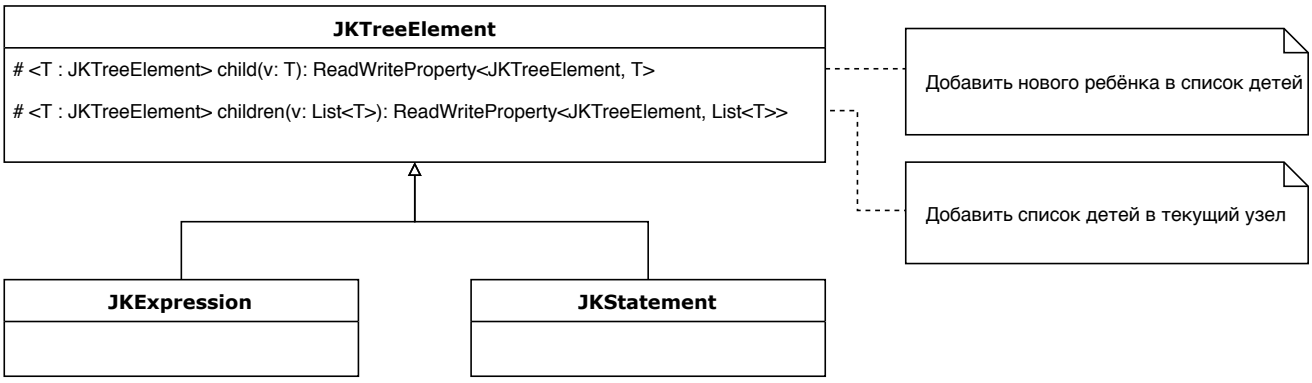

Рис. 2: Структура дерева

AST представлено в виде изменяемого дерева, каждый элемент которого соответствует структуре исходного кода языка Java или Kotlin. Так как дерево изменяемо, то в нём необходимо следить за тем, чтобы какой-нибудь из элементов не был ребёнком сразу двух элементов. Для этого все дети у вершины объявлены с использованием делегатов, которые следят за тем, чтобы у каждого элемента был только один родитель. Общая структура дерева показана на рис. 2. Узлы в дереве бывают трёх типов: конструкции, специфичные только для Java или Kotlin, а также конструкции, общие для обоих языков.

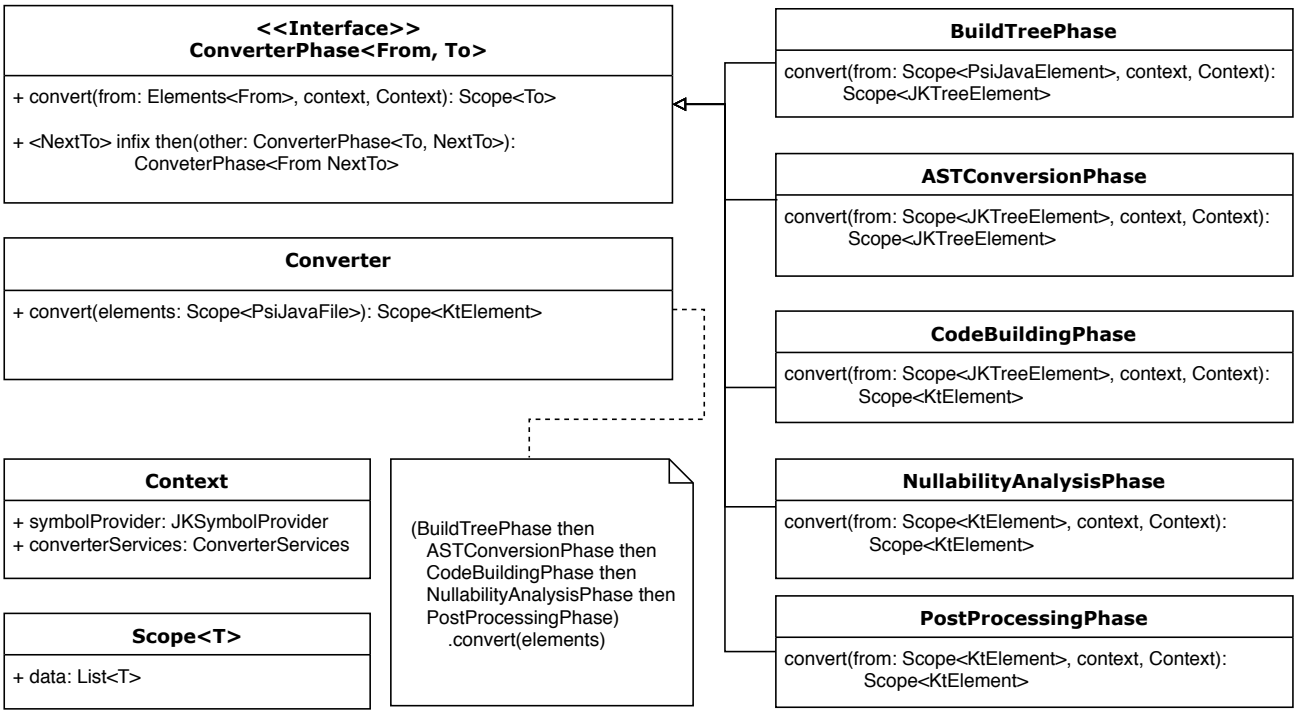

Рис. 3: Диаграмма классов транслятора

3.2. Процесс трансляции

Транслятор состоит из 5 фаз, которые последовательно выполняются друг за другом. Каждая фаза представляет собой выполняющую трансляцию функцию, инкапсулированную в интерфейс `ConveterPhase<From, To>`. Две фазы, подходящие друг другу (то есть, когда у первой фазы тип выходного значения равен типу входного значения второй фазы), могут быть скомбинированы с помощью инфиксной функции `then :(ConverterPhase<To, NextTo>) -> ConveterPhase<From, NextTo>`. Транслятор представляет собой комбинацию всех фаз трансляции, что показано на рис. 3. Здесь `Elements<T>` инкапсулирует в себе элементы типа T, которые должны быть оттранслированы. Для изначальных входных данных это может быть, например, набор Java файлов или фрагментов Java кода. `ConversionContext` представляет из себя информацию, которая необходима для трансляции. Туда входит таблица символов и набор различных сервисов, предоставляемых IntelliJ IDEA, для работы с кодом. Фазы трансляции перечислены ниже.

1. Построение AST по Java коду, представленного в виде Java PSI (`BuildTreePhase`);
2. Выполнение преобразований над полученным AST (`ASTConversionPhase`);
3. Превращение AST в код на языке Kotlin (`CodeBuildingPhase`);
4. Вывод nullability для типов (`NullabilityAnalysisPhase`);
5. Пост-обработка кода (`PostProcessingPhase`).

Рассмотрим подробнее каждую из фаз.

3.2.1. Построение AST по Java коду

Код в IntelliJ IDEA представлен в виде дерева из PSI элементов. Вначале для каждого объявления строится заготовка таблицы символов, а затем рекурсивно строится AST, параллельно заполняя таблицу символов. Полученный AST будет состоять из конструкций, специфичных для языка Java, а также из конструкций, используемых в обоих языках. Также во время данного этапа сохраняется форматирование для элементов.

3.2.2. Выполнение преобразований над AST

На данной фазе происходит превращение Java узлов в соответствующие им Kotlin узлы, которое происходит с помощью последовательного выполнения различных преобразований. Преобразование представляет из себя функцию, которая принимает произвольный узел дерева и возвращает новый (параллельно возможно изменяя существующие узлы). Каждое из преобразований вызывает в конце своей работы самого себя на детях получившегося в результате преобразования узла.

На данный момент в трансляторе около шестидесяти таких преобразований. Некоторые из них достаточно тривиальны (например, преобразования, исправляющие модификаторы видимости и модальности). Некоторые из преобразований представлены ниже.

```
Method("java.util.Collections.singletonList")
    convertTo Method("kotlin.collections.listOf")
    withReplaceType ReplaceType.REPLACE_WITH_QUALIFIER
```

Рис. 4: Описание преобразования

- Преобразование, транслирующее методы из стандартной библиотеки языка Java в соответствующие методы или свойства языка Kotlin. Данное преобразование работает следующим образом. С помощью специального DSL (Domain Specific Language) описывается список преобразований в виде пар: что мы хотим преобразовать (вызов Java метода или доступ к полю Java класса) и во что (в вызов соответствующей функции или обращение к свойству Kotlin класса) с некоторым набором параметров. Например, преобразование из выражения вида `Collections.singletonList(42)` в выражение `listOf(42)` будет записано, как показано на рис. 4.

- Преобразование, добавляющее явные преобразования типов для тех случаев, когда в языке Java они неявные. Например, в языке Java, система типов менее строгая, чем в языке Kotlin, и поэтому в Java выражение `someInt == someChar` является корректным, а в языке Kotlin — нет, эквивалентное утверждение будет выражено как `someInt == someChar.toInt()`. Трансляцией первого в последнее как раз и занимается данное преобразование.

Такой способ разделения обязанностей между преобразованиям, а также то, что преобразования независимы, даёт возможность, во-первых, модифицировать эти преобразования независимо друг от друга, а во-вторых, упрощает добавление новых.

3.2.3. Превращение AST в код на языке Kotlin

Когда уже отсутствуют узлы, представляющие из себя Java код, то дерево превращается в текст на языке Kotlin, путём рекурсивного его обхода с помощью паттерна Visitor. На данной фазе трансляции

также выводятся пробельные символы, собранные на первом этапе, и комментарии для сохранения форматирования исходного файла.

3.2.4. Вывод nullability для типов

На данной фазе происходит вывод nullability для типов. Эта фаза вызывается, когда уже есть код на Kotlin и по нему уже построено PSI дерево, над которым происходят все дальнейшие вычисления. Основная идея данного преобразования состоит в том, что для каждого объявления, имеющего тип (функция, переменная, параметр функции и т.п.) строится, а затем и решается система ограничений, основанная на отношениях вида "тип `A` является подтипом типа `B`", что влечёт что, если тип `A` — nullable, то и, соответственно, тип `B` также nullable или, если тип `B` — not null, то и `A` — not null. Подробнее данный процесс описан в разделе 4.

3.2.5. Пост-обработка кода

В этой фазе трансляции происходит работа с кодом на Kotlin, который представлен в виде PSI дерева. Над этим PSI деревом выполняются следующие виды преобразований:

- вызов инспекций, встроенных в InteliJ IDEA (например, инспекция, удаляющая лишние модификаторы видимости);
- использование различных преобразований над PSI, призванных оптимизировать код, среди которых:
  - конвертация геттеров и сеттеров в свойства языка Kotlin,
  - удаление лишних объявлений типов,
  - сокращение квалифицированных имён типов.

4. Вывод nullability

Рассмотрим множество `N = { nullable, not_null, unknown }`, элементы которого представляют из себя nullability какого-то типа, где `unknown` говорит, что либо мы ещё не успели вычислить nullability, либо не можем его вычислить, так как нам не хватает данных (например, в случае попытки вычислить nullability возвращаемого значения неиспользуемой функции с пустым телом).

Введём на данном множестве частичный порядок с помощью бинарного оператора `<:` следующим образом:

- `not_null <: nullable` (в языке Kotlin не `nullable` тип является подтипом соответствующего ему `nullable` типа)
- $\forall$ `T` $\in$ `N` $\Rightarrow$ `T <: T` (любой тип является подтипом самого себя)
- $\forall$ `T` $\in$ `N` $\Rightarrow$ `T <: unknown` (тип `unknown` в дальнейшем может стать как `nullable`, так и `not_null`)

И так же введём отношение эквивалентности `:=` естественным образом: `T1 := T2` $\iff$ `T1 = T2`

Два данных бинарных отношения создают два вида соответствующих ограничений, которые получаются в процессе анализа кода. У каждого ограничения также есть приоритет, нужный, чтобы убрать неоднозначности при решении (например, ограничение, полученное из инициализатора переменной и говорящее, что она nullable, важнее ограничения, полученного благодаря обращению к полю переменной)

Мы хотим вычислять nullability для следующих типов:

- тип свойств и локальных переменных,
- тип возвращаемого значения функций,
- тип параметров функций,
- тип типовых параметров для функций,

Для каждого типа, объявленного выше, и его типовых параметров создадим типовую переменную. По умолчанию её nullability — это `unknown`.

Типы в языке Kotlin могут иметь типовые параметры (например, тип `List<T>` имеет один типовой параметр `T`). Поэтому также будем при создании системы ограничений учитывать и nullability типовых параметров следующим образом: когда создаём ограничение вида `A <: B` (где `A<T1, ..., Tn>` и `B<S1, ..., Sn>` — некоторые типы с типовыми параметрами `<T1, ..., Tn>`, `<S1, ..., Sn>` для типов `A` и `B` соответственно), мы также для всех пар типовых параметров будем создавать ограничения: пусть `Ti` и `Si` — это типовые параметры с индексом `i` типов `A` и `B` соответственно. Далее рассмотрим три случая.

- Параметры с индексом `i` ковариантные (то есть, если `Ti` — это подтип `Si`, то `A<T1, ..., Ti, ..., Tn>` есть подтип `B<S1, ..., Si, ..., Sn>`). В этом случае создаём ограничение `Ti <: Si`.
- Параметры с индексом `i` контравариантные (то есть, если `Ti` — это надтип `Si`, то `A<T1, ..., Ti, ..., Tn>` есть подтип `B<S1, ..., Si, ..., Sn>`). В этом случае создаём ограничение `Si <: Ti`.
- Параметры с индексом `i` инвариантные (то есть, если `Ti` не равен `Si`, то `A<T1, ..., Ti, ..., Tn>` не находится в отношении "является подтипом" с `B<S1, ..., Si, ..., Sn>`). В этом случае создаём ограничение `Ti := Si`.

4.1. Построение системы ограничений

Для удобства обозначим за `N(x)` nullability выражения `x`. Проанализируем код и найдём использование каждого объявления, для типа которого мы хотим найти nullability:

- `x = y` создает ограничение `N(y) <: N(x)`;

```
val listOfLists: T1@ List<T2@ List<T3@ Int>> =
  emptyList()
val x: T4@ List<T5@ Int> sublilst =
  listOfLists.get(0).sublist(0, 42)
```

Рис. 5: Пример вычисления ограничений для квалифицированной цепочки вызовов

- `x.y` создаёт ограничение `N(x) := not_null`, так как мы обращаемся к члену объекта `x`, что влечёт, что `x != null`;
- проверка на равенство `null`: `x == null` создаёт ограничение `N(x) := nullable`, так как, раз в исходном коде на Java была проверка на `null`, то, скорее всего, подразумевалось, что значение могло быть `null`;
- передача выражения в функцию в качестве параметра: `f(x)` создаёт ограничение `N(x) <: N(p)`, где `p` — это параметр функции `f`.

Выражения, используемые выше, условно можно поделить на 3 типа:

1. Использование объявления, nullability которого мы хотим найти, тогда nullability этого выражения — это просто nullability данного объявления. В примере на рис 5 — это переменные `listOfLists`, `x`;
2. Выражения, никак не зависящие от nullability рассматриваемых объявлений (например, вызов сторонней функции или использование литерала), тогда nullability данного выражения берётся из его типа. В примере на рис 5 — это целочисленные литералы и вызов функции `emptyList`;
3. Квалифицированная цепочка вызовов над исходными объявлениями, например, как показано на рис. 5. В данном случае последовательно рассматривается цепочка выражений слева направо

и для каждого выражения вычисляется его тип с учётом всех вовлечённых типовых параметров. Вычисление происходит с учётом вариантности. В примере, что на рис. 5, – это цепочка вызовов `listOfLists.get(0).sublist(0, 42)`.

Рассмотрим вычисление на примере, показанном на рис. 5. Здесь вспомогательными аннотациями вида `Ti@` помечены типовые переменные для удобства обозначений. `listOfLists` имеет тип `T1@ List<T2@ List<T3@ Int>>`. Далее, так как мы вызываем функцию `get` на переменной `listOfLists`, то сразу получаем первое ограничение `T1 := not_null`. Далее тип выражения `listOfLists.get(0)` будет `T2@ List<T3@ Int>`, а вся правая часть `listOfLists.get(0).sublist(0, 42)` будет иметь тип `List<T3@ Int>`, так как функция `sublist` возвращает новый список с тем же типом (`T3@ Int`), что и в оригинальном списке. Из-за присвоения получим ограничения `T2 <: T4` и `T3 <: T5`. Последнее получим из-за того, что `List` ковариантен по своему типовому параметру. И также получим ограничение `T2 := not_null` из-за вызова метода `sublist`.

4.2. Решение системы ограничений

Для решения полученной системы ограничений воспользуемся следующими правилами:

1. `T <: not_null` $\Rightarrow$ `T := not_null`;
2. `nullable <: T` $\Rightarrow$ `T := nullable`;
3. `T1 <: T2` и мы уже вычислили nullability `T1` $\Rightarrow$ выполняем подстановку `N(T1) <: T2`;
4. `T1 <: T2` и мы уже вычислили nullability `T2` $\Rightarrow$ выполняем подстановку `T1 <: N(T2)`.

В случае, когда есть несколько ограничений, которые мы можем использовать, то выбираем ограничение с наибольшим приоритетом.

В некоторый момент может оказаться, что у нас нет правил, которые мы можем применить. Тогда возможны два варианта: первый — для каждой типовой переменной мы успешно вычислили nullability, второй — это когда есть типовые переменные, для которых nullability остался не найденным. Во втором случае просто установим значение nullability таких типовых переменных в значение по умолчанию: `nullable`.

Проект	Java файлов	Строчек кода	Строчек кода на файл	Новый транслятор			Старый транслятор		
				Ошибок на файл	Всего ошибок	Время работы (сек)	Ошибок на файл	Всего ошибок	Время работы (сек)
Atlassian Jenkins v2.164.2	1557	214282	138	7.1	11057	2109	24.6	38412	977
Badlogicgames LibGDX v1.9.9	580	104841	180	8.1	4673	1373	31.6	18304	529
Google Guava v21.7	572	86944	152	4	2288	1285	35	20020	467
Apache Commons Lang v3.9	154	27889	181	7.9	1217	212	37	5695	118

Таблица 1: Запуск трансляторов на проектах с открытым исходным кодом

5. Апробация

Хотя написанный транслятор и не рассчитан на преобразование проектов целиком, в качестве апробации он был протестирован на нескольких реальных проектах с открытым исходным кодом, написанных на языке Java.

Ввиду того, что старый транслятор генерировал огромное количество ошибок компиляции, в качестве критерия оценки были выбраны метрики: ”количество ошибок компиляции в проекте” и производная метрика от неё: ”количество ошибок компиляции на файл”. Также было измерено время работы для каждого из трансляторов. Для каждого из проектов было измерено количество Java-файлов в нём, а также количество строчек кода в проекте, которое было замерено с помощью утилиты SLOCCount [13]. Результаты апробации представлены в таблице 1.

Анализируя данные результаты, можно сделать вывод, что новый транслятор генерирует код с количеством ошибок компиляции меньшим в 5 раз в сравнении со старым, но в тоже время, в 2.5 раз медленнее работает.

6. Заключение

В рамках данной работы были получены следующие результаты:

- Проанализированы недостатки старой версии транслятора, которыми являются:
  - Генерация кода на языке Kotlin с большим количеством ошибок компиляции;
  - Отсутствие расширяемости и сложность модификации преобразований.
- Реализована новая версия транслятора из Java в Kotlin для среды IntelliJ IDEA.
- Проведено сравнение нового транслятора со старым и получено, что:
  - Новая версия транслятора в среднем в 5 раз генерирует меньше ошибок компиляции, нежели старая;
  - Новая версия транслятора более расширяемая, чем старая.

Исходный код данной работы находится на Github[1]. Работа выполнялась под именем пользователя darthorimar.

[1] https://github.com/jetbrains/kotlin

Список литературы

[1] Compilers: Principles, Techniques, and Tools (2Nd Edition) / Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2006. — ISBN: 0321486811.

[2] Converting a Java File to Kotlin File - Help | IntelliJ IDEA. — 2019. — Access mode: `https://www.jetbrains.com/help/idea/converting-a-java-file-to-kotlin-file.html` (online; accessed: 2019-03-01).

[3] Damas Luis, Milner Robin. Principal Type-schemes for Functional Programs // Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '82. — New York, NY, USA : ACM, 1982. — P. 207–212. — Access mode: `http://doi.acm.org/10.1145/582153.582176`.

[4] External annotations - Help | IntelliJ IDEA. — 2019. — Access mode: `https://www.jetbrains.com/help/idea/external-annotations.html` (online; accessed: 2019-03-01).

[5] Hubert Laurent. A Non-null Annotation Inferencer for Java Bytecode // Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. — PASTE '08. — New York, NY, USA : ACM, 2008. — P. 36–42. — Access mode: `http://doi.acm.org/10.1145/1512475.1512484`.

[6] IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains. — 2019. — Access mode: `https://www.jetbrains.com/idea/` (online; accessed: 2019-03-01).

[7] Knuth Donald E. Semantics of context-free languages // Mathematical systems theory. — 1968. — Jun. — Vol. 2, no. 2. — P. 127–145. — Access mode: `https://doi.org/10.1007/BF01692511`.

[8] Kotlin Programming Language. — 2019. — Access mode: https://kotlin (online; accessed: 2019-03-01).

[9] Kotlin is now Google's preferred language for Android app development | TechCrunch<. — 2019. — Access mode: https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-develo (online; accessed: 2019-04-08).

[10] Null Safety - Kotlin Programming Language. — 2019. — Access mode: https://kotlinlang.org/docs/reference/null-safety.html (online; accessed: 2019-03-01).

[11] @Nullable and @NotNull - Help | IntelliJ IDEA. — 2019. — Access mode: https://jetbrains.com/help/idea/nullable-and-notnull-annotations.html (online; accessed: 2019-03-01).

[12] Program Structure Interface (PSI) | IntelliJ Platform SDK DevGuide. — 2019. — Access mode: https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html (online; accessed: 2019-03-01).

[13] SLOCCount. — 2019. — Access mode: https://dwheeler.com/sloccount (online; accessed: 2019-03-01).