УДК 004.4'422
DOI 10.25205/1818-7900-2020-18-2-15-30

Разработка внутреннего представления компилятора Kotlin / Native и оптимизаций на его основе

С. С. Боголепов

Новосибирский государственный университет
Новосибирск, Россия

Аннотация

Котлин – это статически типизированный язык программирования, который поддерживает объекно-ориентированную и функциональную парадигмы программирования. Изначальной целевой платформой была выбрана JVM, однако затем была добавлена возможность транспиляции в JavaScript и компиляции под нативные платформы с помощью LLVM (Kotlin / Native). Первые две платформы представляют собой хорошо развитые виртуальные машины, способные выполнять продвинутую оптимизацию программ во время исполнения. Однако в случае нативных платформ оптимизацию необходимо выполнять во время компиляции.

На данный момент в Kotlin / Native отсутствуют многие оптимизации, из-за чего производительность порождаемого кода во многих случаях получается низкой. В этой работе описан способ решения данной проблемы с помощью введения дополнительного внутреннего представления, основанного на SSA-форме, и реализации escape-анализа на его основе. Результаты экспериментов показали, что этот подход способен значительно улучшить производительность.

Ключевые слова

Kotlin, static single assignment, escape-анализ, оптимизация, LLVM

Для цитирования

Боголепов С. С. Разработка внутреннего представления компилятора Kotlin / Native и оптимизаций на его основе // Вестник НГУ. Серия: Информационные технологии. 2020. Т. 18, № 2. С. 15–30. DOI 10.25205/1818-7900-2020-18-2-15-30

Development of Kotlin / Native Intermediate Representation and Optimizations

S. S. Bogolepov

Novosibirsk State University
Novosibirsk, Russian Federation

Abstract

Kotlin is a statically typed programming language that supports object-oriented and functional programming. It supports JVM, JS and native platforms via LLVM (Kotlin / Native). The first two targets are backed with well-developed virtual machines that can perform advanced program optimizations at runtime. However, for native platforms, all optimizations must be performed at compile time.

Currently Kotlin / Native lacks many optimizations, which is why the performance of the generated code is poor in many cases. This paper describes a way to solve this problem by introducing an additional SSA-based intermediate representation and implementing escape analysis using it. Experimental results have shown that this approach can significantly improve performance.

Keywords

Kotlin, static single assignment, escape analysis, optimization, LLVM

For citation

Bogolepov S. S. Development of Kotlin / Native Intermediate Representation and Optimizations. *Vestnik NSU. Series: Information Technologies*, 2020, vol. 18, no. 2, p. 15–30. (in Russ.) DOI 10.25205/1818-7900-2020-18-2-15-30

Введение

Kotlin / Native – это AOT (Ahead-of-Time) компилятор языка программирования Котлин. Для компиляции в машинный код используется LLVM. Поддерживается большое количество целевых платформ, таких как iOS, macOS, Linux (arm64, x64, mips32) и многие другие.

Примечательной особенностью Kotlin / Native является то, что он компилируется в режиме «закрытого мира», что отличает его от большинства других промышленных компиляторов. Режим «закрытого мира» означает, что во время компиляции доступен весь граф зависимостей текущей единицы трансляции, что позволяет делать глобальные анализ и оптимизацию программы. Например, известна вся иерархия классов, что упрощает оптимизацию виртуальных вызовов.

Недостатком по сравнению с «открытым миром» является усложнение раздельной компиляции, а также трудности с одновременным использованием нескольких динамических библиотек, написанных на Kotlin / Native. Например, это приводит к тому, что стандартная библиотека и другие зависимости включены в каждый бинарный файл. Если одна зависимость используется в двух разных бинарных файлах, то это разные с точки зрения идентичности декларации, так как они принадлежат к разным «мирам».

Внутреннее представление компилятора Kotlin / Native

Внутреннее (или промежуточное) представление (Intermediate Representation, IR) компилятора – структура данных, используемая компилятором для представления исходного кода входной программы. Многие компиляторы используют несколько внутренних представлений на разных этапах компиляции, так как некоторые алгоритмы трансляции и оптимизации удобнее реализовывать с помощью одного представления, а другие – с помощью другого.

Рассмотрим, как эволюционировал компилятор Котлина с точки зрения используемых внутренних представлений.

Program Structure Interface. Изначально в компиляторе Котлина не было продвинутого внутреннего представления, пригодного для оптимизаций. PSI (Program Structure Interface, представление кода, использующееся в Intellij Platform) напрямую транслировался в байткод виртуальной машины Java. Такое же решение использовалось и при трансляции в JavaScript. Данный подход был приемлем, так как в обоих случаях задача оптимизации кода программы целиком ложилась на плечи виртуальной машины. Тем не менее даже для таких примитивных целей (с точки зрения продвинутых компиляторов) это не самый удачный выбор, так как, например, подстановку инлайн-функций приходится выполнять на байткоде JVM, что является крайне нетривиальной задачей. Поэтому вместе с разработкой Kotlin / Native началась разработка полноценного нового внутреннего представления, общего для всех компиляторов.

Древовидное представление. Выбор пал на древовидное представление программ. Оно низкоуровневое, по сравнению с PSI, изменяемое и сериализуемое, что позволяет использовать его для реализации оптимизаций и в качестве формата для дистрибуции библиотек. В отличие от многих других компиляторов, линковка в компиляторе Kotlin / Native происходит на уровне промежуточного представления (а не бинарного кода), поэтому на вход оптимизирующим фазам компилятора подается вся программа с ее зависимостями. Разумеется, это дорогой с точки зрения времени компиляции подход, поэтому для тех режимов компиля-

ции, в которых не важна производительность порождаемого кода, существует кэширование зависимостей в виде бинарного кода.

Над древовидным представлением можно производить понижающие преобразования, что позволяет оптимизировать программный код и упрощает дальнейшую трансляцию. Но так как отсутствует явный граф потока управления, реализация многих оптимизаций, основанных на нем, затруднена. Поэтому, например, оптимизация циклов выполнена ad hoc для известных коллекций из стандартной библиотеки, что существенно ограничивает область ее применения.

Рассмотрим пример. Пусть на вход компилятору дана следующая функция f:

```
@SSA
fun f(x: Boolean): Int {
    val a = if (x) {
        A(5)
    } else {
        A(6)
    }
    a.print()
    return 0
}
```

Тогда текстовое описание ее древовидного представления выглядит так:

```
FUN name:f visibility:public modality:FINAL <>
(x:kotlin.Boolean) returnType:kotlin.Int
      annotations:
        SSA
      VALUE_PARAMETER name:x index:0 type:kotlin.Boolean
      BLOCK_BODY
        VAR name:a type:<root>.A [val]
          WHEN type=<root>.A origin=IF
            BRANCH
              if: GET_VAR 'x: kotlin.Boolean'
type=kotlin.Boolean
              then: BLOCK type=<root>.A
                CONSTRUCTOR_CALL 'public constructor <init>
(arg: kotlin.Int) [primary]' type=<root>.A
                  arg: CONST Int type=kotlin.Int value=5
            BRANCH
              if: CONST Boolean type=kotlin.Boolean value=true
              then: BLOCK type=<root>.A origin=null
                CONSTRUCTOR_CALL 'public constructor <init>
(arg: kotlin.Int) [primary]' type=<root>.A
                  arg: CONST Int type=kotlin.Int value=6
        CALL 'public final fun print (): kotlin.Unit'
type=kotlin.Unit
          $this: GET_VAR 'val a: <root>.A [val]' type=<root>.A
        RETURN type=kotlin.Nothing
          CONST Int type=kotlin.Int value=0
```

LLVM

LLVM (Low-Level Virtual Machine) – это инфраструктура разработки компиляторов [1]. В ее основе лежит низкоуровневое внутреннее представление, основанное на графе потока управления в SSA-форме – LLVM IR (сериализованная форма которого называется «биткод»). Фронтенду компилятора достаточно транслировать свое представление (например, древовидное) в LLVM IR, после чего LLVM выполнит его оптимизацию и компиляцию под целевую платформу. Оптимизации LLVM не привязаны к какому-то конкретному языку и поэтому достаточно низкоуровневые. Пример оптимизаций, которые поддерживает LLVM:

- mem2reg – замена операций работы с памятью на операции работы с виртуальными регистрами;
- Global Value Numbering – поиск общих подвыражений;
- удаление мертвого кода (как целых функций, так и инструкций и базовых блоков).

LLVM лег в основу большого количества современных компиляторов: Clang, Rust, Swift, Julia, Crystal, Scala Native и многих других.

Static Single Assignment (SSA) форма

Промежуточное представление программы, в котором каждое значение присваивается ровно один раз. Данное представление значительно упрощает написание многих видов оптимизаций, так как отсутствуют изменяемые переменные. Важной особенностью данного представления является φ-функция, которая служит для слияния нескольких значений в одно в случае нескольких предшествующих ветвей управления (рис. 1, 2).

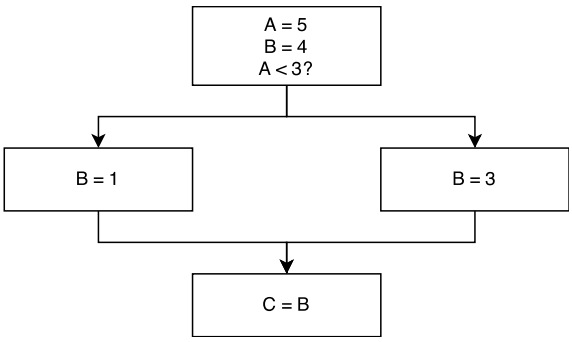

Рис. 1. Пример графа потока управления не в SSA-форме
Fig. 1. Non-SSA Control Flow Graph

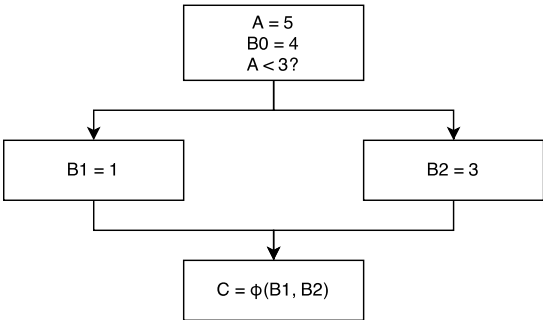

Рис. 2. Пример того же графа управления в SSA-форме
Fig. 2. The same Control Flow Graph in SSA form

Управление памятью

В отличие от большинства реализаций JVM и виртуальных машин JS, в которых для сборки мусора применяется трассирующий коллектор, в Kotlin / Native используется автоматический подсчет ссылок (рис. 3). Наивный алгоритм подсчета ссылок плохо применим к языку Котлин, так как он приводит к утечкам памяти, поэтому используется модификация алгоритма, описанного в работе [2].

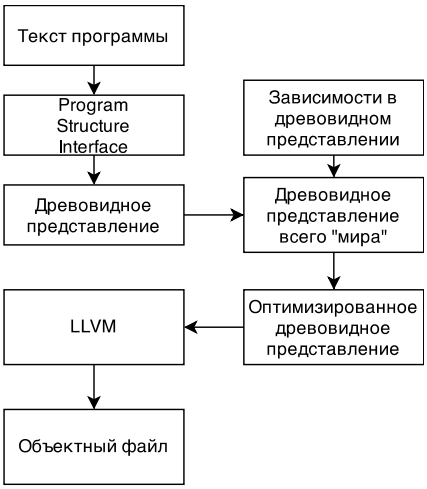

Рис. 3. Текущий пайплайн компилятора Kotlin / Native
Fig. 3. Current Kotlin / Native compiler pipeline

Одной из основных причин для выбора данного алгоритма была необходимость взаимодействия со средой исполнения Objective-C, где также используется подсчет ссылок.

Замеры текущей производительности

Рассмотрим сравнение производительности разных языков [1]. Оно нас интересует из-за того, что в тестовой программе много аллокаций объектов и мало виртуальных вызовов, что позволяет оценить влияние управления памятью на общую производительность кода. Рассмотрим результаты измерения профиля программы с помощью утилиты perf, приведенные в (табл. 1). Как видно из таблицы, все указанные функции (за исключением splitBinary и merge) относятся к управлению памятью.

Данная компонента среды исполнения является бутылочным горлышком Kotlin / Native в плане производительности, так как стиль программирования на языке Котлин предполагает большое количество объектов и присвоения ссылок, что приводит к большому количеству вызовов функций управления памятью. В отличие от трассирующего коллектора, который используется в подавляющем большинстве сред исполнения с автоматической сборкой мусора, в средах исполнения с подсчетом ссылок каждое создание объекта и изменение ссылки приводит к замедлению программы.

[1] https://github.com/frol/completely-unscientific-benchmarks

Таблица 1

Профиль в Unscientific Benchmark

Table 1

Unscientific Benchmark profile

Функция и ее происхождение	Доля процессорного времени
garbageCollect (управление памятью)	19,7
allocInstance (управление памятью)	16
kfun:splitBinary (пользовательский код)	10,1
freeContainer (управление памятью)	10
updateHeapRef (управление памятью)	6,3
kfun:merge (пользовательский код)	5,5
ReleaseHeapRefStrict (управление памятью)	5,3
Остальные функции	27,1

Кроме того, ошибочно воспринимать LLVM как JVM, CLR и другие виртуальные машины. Несмотря на то, что фундаментально данные технологии похожи (трансляция и оптимизация некоторого низкоуровневого входного языка в бинарный код), концептуально они совершенно разные: оптимизирующий пайплайн LLVM ожидает на вход достаточно примитивный язык и не умеет оптимизировать паттерны, соответствующие коду, написанному на объектно-ориентированном (ОО) языке, в то время как, например, JVM изначально создавалась с расчетом, что ей на вход подается байткод, который получен трансляцией программы на языке Java практически без оптимизаций.

Возможные решения

Так как нас интересует уменьшение доли вышеперечисленных функций в профиле программы, то возможны два ортогональных подхода:

- динамический – ускорение исполнения путем ручной оптимизации данных функций;
- статический – уменьшение количества вызовов к ним путем компиляторных оптимизаций.

Первый подход более простой, так как не требует внесения изменений в код компилятора, но является ограниченным, поскольку вызовы функций среды исполнения все равно остаются. Кроме того, он никак не улучшает код, порождаемый компилятором. Второй подход потенциально позволяет избавиться вообще от всех операций инкремента / декремента счетчика ссылок в некоторых случаях, но при этом сложнее в реализации.

Для реализации второго подхода часто (в частности, в JVM) используется алгоритм escape-анализа. Цель данного анализа – определить область видимости объекта. Например, если он виден только внутри функции, то его можно аллоцировать на стеке, а если он присваивается в глобальную переменную, то это будет некорректно, и его нужно аллоцировать на куче.

Как и с любым алгоритмом, для реализации escape-анализа очень важно правильное представление входных данных. В данном случае таковым является внутреннее представление компилируемой программы. Рассмотрим, почему ни одно из двух существующих представлений компилятора Kotlin / Native (древовидное и LLVM IR) не являются оптимальными для реализации escape-анализа.

На древовидном IR такую оптимизацию полноценно реализовать оказывается затруднительно, так как отсутствуют явное представление потока управления и операции подсчета

ссылок. В Kotlin / Native на данном представлении реализован алгоритм локального escape-анализа, который крайне консервативен (поддерживает только локальные массивы примитивных типов фиксированной длины).

В LLVM существует развитая инфраструктура для написания анализов и трансформаций, поэтому возникает идея переиспользовать ее для написания собственных. Однако это довольно нетривиально, так как в LLVM отсутствует любая информация о статических типах объектов (например, информация о наследовании), которая необходима для написания высокоуровневых оптимизаций. Ее можно передать в виде метаинформации, но работать с ней в таком случае крайне затруднительно.

Таким образом, возникает идея для добавления еще одного внутреннего представления, которое, с одной стороны, будет знать об особенностях входного языка, а с другой – отражать поток управления и операции подсчета ссылок. Похожие решения применяются во многих других компиляторах, которые используют LLVM (например, Swift, Rust и Scala Native).

В дальнейшем новое внутреннее представление будет именоваться SSA IR (Static-Single Assignment Intermediate Representation).

Описание реализации

SSA IR – это внутреннее представление, основанное на графе потока управления, находящегося в SSA-форме. Рассмотрим основные составляющие данного представления.

Модуль. Это коллекция определений функций и глобальных переменных. На данный момент модуль выполняет единственную роль: это удобный способ запустить какую-то трансформацию или анализ над всеми элементами SSA IR.

Функция. Это именованный направленный граф базовых блоков, соединенных между собой параметризованными ребрами. Параметры функции совпадают с параметрами входного базового блока. У методов классов и интерфейсов нет неявного параметра this, он указывается явно первым параметром.

Для каждой функции хранится ссылка на соответствующую декларацию древовидного представления, из которой можно получить дополнительную информацию (например, область видимости функции).

Базовый блок. Это последовательность инструкций, у которой одна точка входа (первая инструкция) и одна точка выхода (терминальная инструкция). Одной из особенностей реализации является то, что вместо классических φ-функций используются параметры у базовых блоков. Данный подход, во-первых, делает взаимосвязи между базовыми блоками более очевидными, а во-вторых, решает ряд проблем классического подхода. Например:

- φ-функции всегда должны быть в начале базового блока, что усложняет оптимизации;
- невозможность провести два ребра от одного блока к другому.

Кроме того, базовые блоки с параметрами фактически стирают границу между SSA-представлением и Continuation Passing Style [3], что упрощает процесс решения проблем при использовании подобного представления.

Инструкция. Аналогично LLVM IR инструкции представлены в виде трехадресного кода (т. е. имеют явно указанное возвращаемое значение и параметры).

В отличие от LLVM IR, набор инструкций SSA IR позволяет описывать более высокоуровневые операции. Рассмотрим несколько примеров.

- SSAInterfaceCall. Вызов интерфейсного метода.
- SSAVirtualCall. Вызов виртуального метода.
- SSAIncRef/SSADecRef. Инкремент / декремент счетчика ссылок у переданного объекта.
- SSAAlloc. Абстрактная аллокация (на куче или на стеке).

Для того чтобы упростить работу, была добавлена аннотация @SSA, которая означает, что над аннотированной функцией необходимо проводить нижеописанные операции. Таким образом, стало возможным разрабатывать SSA IR постепенно, аннотируя только те функции, конструкции которых поддержаны в представлении (рис. 4).

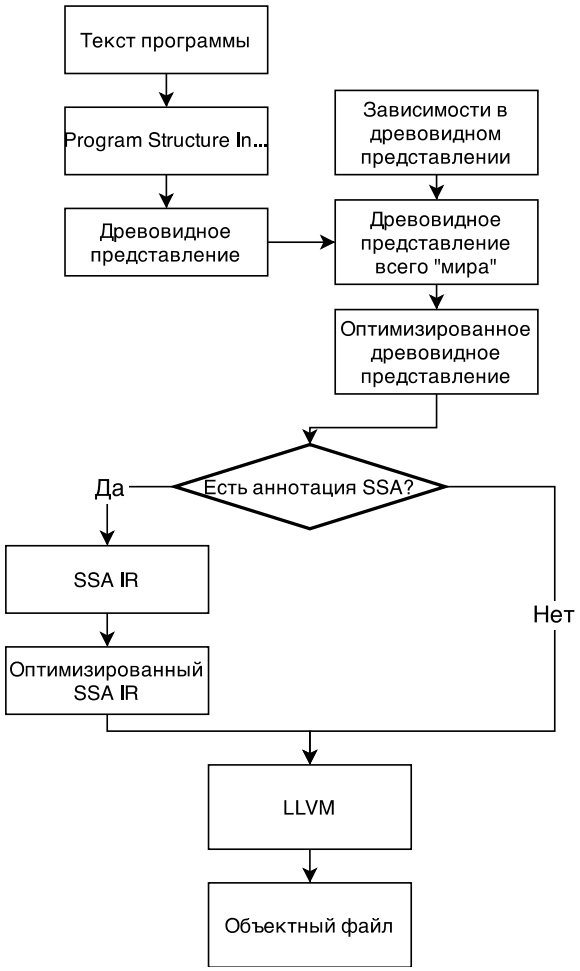

Рис. 4. Новый пайплайн компилятора
Fig. 4. New compiler pipeline

Трансляция в SSA IR

Алгоритм трансляции древовидного представления в SSA IR основан на методе, описанном в работе [4]. В отличие от классического алгоритма, описанного в [5], в данном алгоритме не требуется предварительно строить граф потока управления и фронт доминаторов, что значительно упрощает трансляцию напрямую из древовидного представления.

Алгоритм трансляции

Инструкции древовидного представления, которые выполняются последовательно, оказываются в одном базовом блоке. Когда встречается запись в переменную, мы записываем промежуточное представление, являющееся значением переменной, как *текущее значение пере-*

менной. Соответственно, когда происходит чтение значения переменной, мы его используем. Данный процесс называется *local value numbering.* Целиком заполненный базовый блок помечается как *заполненный.*

Если базовый блок не содержит определения переменной, то мы рекурсивно ищем его среди предков блока. Если предков несколько, то на ребра каждого из них добавляется значение переменной, а базовому блоку добавляется параметр, формальное значение которого указывается в качестве значения переменной. Как можно заметить, при использовании такой нотации (вместо φ-функции) возникает ассоциация с вызовом функции, что значительно упрощает ментальную модель данного представления.

В случае циклов внутри функции может возникнуть рекурсивное определение значения переменной. Поэтому сначала добавляется формальный параметр базового блока, и если при поиске значения переменной мы встречаем формальный параметр, то это означает конец рекурсии.

Поиск значения осуществляется только внутри *заполненных* блоков, так как в незаполненный блок может добавиться новое значение, которое перезапишет предшествующее.

Назовем базовый блок *запечатанным*, если известны все его предки. Стоит отметить, что свойство *запечатанности* не влечет за собой *заполненность* (тривиальный пример – входной базовый блок).

Проблема возникает, когда мы пытаемся найти определение в *незапечатанном* блоке, в котором нет определения этой переменной. В таком случае мы создаем дополнительный параметр блока и указываем его в качестве значения. Для каждого блока мы храним список таких параметров. В дальнейшем, когда происходит *запечатывание* базового блока, для каждого такого параметра осуществляется поиск по алгоритму, описанному ранее.

Полезным побочным эффектом данного алгоритма является то, что ряд оптимизаций происходит прямо во время построения представления. Например, протяжка констант.

Ранее рассмотренная функция f в SSA IR принимает следующий вид:

```
f(%0: bool): (bool) -> int
block entry0():
  condbr %0: bool when_body1() when_cond2()

block when_body1():
  %2 A = allocate
  %3 A = call_direct A.<init> %2: A, 5: int
  br when_exit3(%2: A)

block when_cond2():
  %5 A = allocate
  %6 A = call_direct A.<init> %5: A, 6: int
  br when_exit3(%5: A)

block when_exit3(%8: A):
  %9 type_unk = call_direct A.print %8: A
  br return_block4(0: int)

block return_block4(%11: int):
  ret %11: int
```

Алгоритмы над SSA IR

Все алгоритмы выполнены в форме проходов над представлением, которые запускаются последовательно.

Валидация проверяет инварианты, общие для всех промежуточных состояний представления, а именно:

- все базовые блоки замкнуты;
- все базовые блоки заканчиваются терминальными инструкциями;
- все использования некоторого значения указаны в списке пользователей данного значения;
- инструкция находится внутри базового блока, который указан как ее владелец;
- количество и тип параметров базового блока соответствует количеству и типу аргументов, переданных в него.

Удаление недостижимых блоков. В результате трансляции из древовидного представления или в ходе какого-то преобразования может возникнуть базовый блок, у которого нет предков. Поток управления никак не может прийти в такой базовый блок, поэтому он может быть удален. Данный алгоритм реализован в LLVM, поэтому он не является обязательным. Тем не менее эта оптимизация уменьшает время работы дальнейших алгоритмов и упрощает отладку.

Открытая подстановка геттеров и сеттеров. В Котлине вместо методов вида *getField*() и *setField*() используются *свойства* (property), для которых автоматически генерируются соответствующие методы доступа к полям объекта: геттеры и сеттеры. Их можно переопределять, усложняя поведение, но в большинстве случаев в них нет никакой произвольной логики, кроме обращения к подлежащему полю. Возможность «увидеть» доступ к полям объекта крайне важен для escape-анализа, так как если доступ к полю скрыт внутри метода, то escape-анализ сможет увидеть, что объект *убегает*, через присваивание в поле только на этапе межпроцедурного анализа. Поэтому необходима трансформация, которая осуществляет открытую подстановку (инлайн) таких методов.

Полноценная реализация такой оптимизации является крайне трудоемкой задачей с огромным числом эвристик. Для наших же целей достаточно реализовать простой алгоритм, который подставляет только геттеры и сеттеры с примитивными телами. Компиляция в режиме «закрытого мира» позволяет проанализировать тело вызываемого метода и принять решение о подстановке.

Построение графа вызова функций позволяет выделить компоненты сильной связности (т. е. взаимную рекурсию) и понять, какие функции «скрываются» за виртуальными вызовами. Без него крайне затруднительно проводить межпроцедурные оптимизации.

В компиляторах ООП языков наиболее распространены два алгоритма: Class Hierarchy Analysis и Rapid Type Analysis [6]. В данной работе использована упрощенная версия последнего алгоритма.

Рассмотрим пример работы алгоритма. Пусть на вход компилятору подается следующая программа:

```
interface I {
    fun virtualFn()
}

class A : I {
    override fun virtualFn() {}
}

class B : I {
    override fun virtualFn() {}
}
```

```
class C : I {
    override fun virtualFn() {}
}

@SSA
fun f(isA: Boolean): I = if (isA) A() else B()

@SSA
fun main() {
    f(true).virtualFn()
}
```

В этой программе объект класса C никогда не создается, поэтому в графе вызовов C.virtualFn отсутствует (рис. 5).

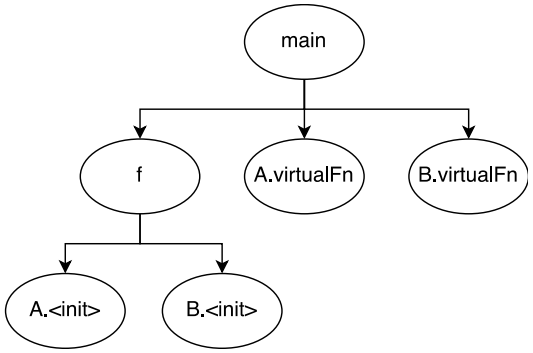

Рис. 5. Пример графа вызовов
Fig. 5. Call graph example

Escape-анализ

Чтобы уменьшить нагрузку на сборщик мусора, необходимо статически вывести время жизни объектов. Те объекты, время жизни которых не превышает время жизни стекового фрейма, можно аллоцировать на стеке. Алгоритм, который реализует такой анализ, называется «escape-анализ». Существует несколько реализаций такого алгоритма. Большинство из них было написано для JVM, которая JIT-компилирует код в режиме открытого мира. Это требует ряда компромиссов, так как в данном случае компилятор разделяет ресурсы компьютера с исполняемым кодом. В нашем же случае компиляция происходит AOT в режиме закрытого мира, что позволяет получить более точные результаты анализа ценой времени компиляции.

Описание работы использованного алгоритма

В данной работе был реализован алгоритм, описанный в статье [7]. Как и многие глобальные анализы, выбранный алгоритм разбивается на две фазы: внутрипроцедурного и межпроцедурного анализа.

Внутрипроцедурный анализ. Эта часть анализа заключается в построении локального графа связей для всех анализируемых функций.

Граф связей. Структура данных, которая используется для вычисления и хранения информации о том, на какие объекты могут ссылаться переменные, поля объектов и параметры функций. Граф связей – это направленный граф (N, E).

N – это множество вершин, которое разделяется на два класса. N_o – множество *объектных* вершин, представляющих объекты. N_r – множество *ссылочных* вершин, представляющих ссылки на объекты. Существует несколько видов ссылочных вершин:

- *локальные* представляют локальные переменные, ссылающиеся на объекты.
- *актуальные* (actual) – параметры метода, аргументы и возвращаемые значения.
- *поля* – поля ссылочного типа.
- *глобальные* – глобальные переменные.

Стоит отметить, что одна объектная вершина может представлять множество вершин времени исполнения (например, если аллокация происходит в цикле).

E – это множество ребер нескольких видов:

- *указатель* – ребро от вершины-ссылки к вершине-объекту;
- *поле объекта* – ребро от вершины-объекта *o* к вершине-ссылке *f*. Существует, только если *f* представляет поле объекта *o*;
- *отложенное* ребро – ребро между двумя ссылками, которое возникает в случае присваивания.

Такие ребра можно удалять с помощью операции *ByPass(q)*. Она заключается в «перекидывании» отложенных ребер, указывающих на *q*, на вершины, на которые указывает *q* (рис. 6).

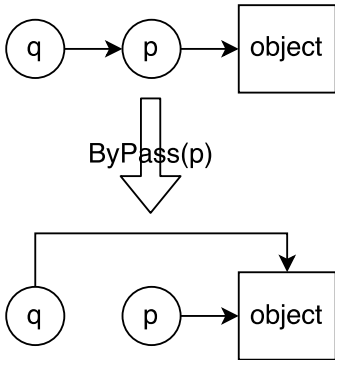

Рис. 6. Пример операции *ByPass(p)*
Fig. 6. ByPass(p) sample

С каждой вершиной ассоциируется одно из двух состояний: *не убегает* или *убегает*. Последнее состояние назначается вершинам, которые живут дольше, чем стековый фрейм анализируемого метода. Состояние *убегает* «растекается» по ребрам, и, таким образом, все транзитивное замыкание «убегающей» вершины становится «убегающим».

Построение графа связей. Граф связей строится последовательным обходом графа потока управления. Циклы обрабатываются до тех пор, пока алгоритм не сойдется. Если спустя фиксированное число итераций алгоритм не сходится, всем вершинам метода консервативно выставляется состояние *убегает*. Граф связей на входе в базовый блок – это объединение всех графов связей его предшественников.

Перед построением графа связей для входного базового блока нужно обработать параметры функции в виде $f = a$, где f – это локальная вершина ссылочного типа, a – актуальная вершина ссылочного типа, которая представляет собой фактический аргумент. О том, как обрабатываются присваивание, рассказано ниже.

Обработка инструкций в базовом блоке.

- Присваивание объекта *O* в вершину *v*. Создается новая объектная вершина для *O*. Затем вызывается *ByPass(v)*, после чего добавляется ребро-указатель от *v* к *O*.

- Присваивание переменной *p* значения переменной *q*. Применяем *ByPass(p)*, затем добавляем *отложенное* ребро от *p* в *q*.
- *p.f* = *q*. Если в графе нет объектных вершин, достижимых из *p* через *указатели* или *отложенные* ребра (например, это параметр функции), то создаем фантомный объект *O* и добавляем ребро-указатель. Потом убеждаемся, что для всех таких объектных вершин существует вершина для поля *f*. Затем для всех *f* добавляем отложенное ребро к *q*. Если *p.f* – это глобальная переменная, то у *q* устанавливается состояние *убегает*.
- *p* = *q.f*. Сначала делаем *ByPass(p)*. Если у *q* нет объектов, на которые она указывает, создаем фантомный объект и добавляем *ребро-указатель*. Создаем ребра и вершины для всех *f*. Затем добавляем *отложенные* ребра от *p* ко всем *f*.

На выходе из метода возвращаемые значения обрабатываются как присваивания в актуальные ссылки. Затем применяется *ByPass* ко всем ссылочным вершинам, чтобы избавиться от всех отложенных ребер. После этого та часть подграфа, которая достижима из *убегающих* вершин формирует *нелокальный подграф*, который представляет собой то, как функция влияет на вызывающую его функцию.

Межпроцедурная часть анализа заключается в уточнении графа вызывающей функции с помощью результатов анализа вызываемой функции.

Аргументы функций рассматриваются как присваивания в *актуальные* ссылочные вершины, т. е. происходит обработка присваивания *a* = *p*, где *a* – это *актуальная ссылка*, которая позднее будет связана с актуальной ссылкой в вызываемой функции (мы ее создаем в самом начале алгоритма).

Сразу после вызова функции мы должны обработать эффект, который вызываемая функция оказывает на вызывающую. Для этого мы должны построить соответствие между двумя графами связи. Мы это делаем начиная с *актуальных* ссылок, которые представляют формальные параметры метода на стороне вызываемой функции с соответствующими ссылками в вызывающей функции. Затем ищем соответствующие объекты в множествах *PointsTo*. Если в вызывающей функции нет соответствующей вершины для объекта, то создается фантомный объект, после чего строим соответствие между полями данных объектов. Этот процесс продолжается рекурсивно до полного замыкания. Далее добавляем соответствующие ребра по аналогичному правилу.

Таким образом, *нелокальный подграф* вызываемой функции копируется в граф вызывающей функции, и его эффект распространяется на последнюю.

Возникает естественный вопрос о рекурсивных вызовах. Для этого выделяются компоненты сильной связности, и межпроцедурный анализ запускается на них до тех пор, пока алгоритм не сойдется.

Расстановка операций управления памятью. Следующая фаза использует результат escape-анализа для расстановки операций инкремента и декремента счетчика ссылок.

В случае если объект не покидает пределов метода, он просто аллоцируется на стеке и помечается специальной маской, чтобы сборщик мусора не удалил такой объект, несмотря на отсутствие живых ссылок на него.

Иначе происходит расстановка операций изменения счетчика ссылок:

- при передаче объекта в качестве аргумента в функцию счетчик увеличивается;
- после возвращения из функции счетчик ссылок уменьшается;
- при записи в поле счетчик ссылок увеличивается, а у объекта, на который до этого указывало поле, уменьшается.

Трансляция из SSA IR в LLVM IR

Так как трансляция в LLVM осуществляется из низкоуровневой вариации SSA IR, то данный процесс становится тривиальным. Параметры базовых блоков транслируются в φ-функции, а большинство остальных инструкций SSA IR – в их аналоги в LLVM. Исключение

составляют операции, которые не отражены в LLVM. Они транслируются в вызовы функций среды исполнения.

Функция f в результате трансляции в LLVM IR выглядит следующим образом. Как видно по инструкции alloca, объект класса A аллоцируется на стеке:

```
define i32 @"kfun:#f(kotlin.Boolean){}kotlin.Int"(i1) #11 {
entry:
  br i1 %0, label %when_body, label %when_cond

when_body:                                        ; preds = %entry
  %1 = alloca %"kclassbody:A#internal"
  %2 = bitcast %"kclassbody:A#internal"* %1 to i8*
  call void @llvm.memset.p0i8.i32(i8* %2, i8 0, i32 16, i1 false)
  %3 = bitcast %"kclassbody:A#internal"* %1 to %struct.ObjHeader*
  %typeInfoOrMeta  = getelementptr inbounds %struct.ObjHeader,
%struct.ObjHeader* %3, i32 0, i32 0
  store %struct.TypeInfo* @"kclass:A", %struct.TypeInfo** %typeInfoOrMeta_
  call void @"kfun:A#<init>(kotlin.Int){}"(%struct.ObjHeader* %3, i32 5)
  br label %when_exit

when_cond:                                        ; preds = %entry
  %4 = alloca %"kclassbody:A#internal"
  %5 = bitcast %"kclassbody:A#internal"* %4 to i8*
  call void @llvm.memset.p0i8.i32(i8* %5, i8 0, i32 16, i1 false)
  %6 = bitcast %"kclassbody:A#internal"* %4 to %struct.ObjHeader*
  %typeInfoOrMeta_1 = getelementptr inbounds %struct.ObjHeader,
%struct.ObjHeader* %6, i32 0, i32 0
  store %struct.TypeInfo* @"kclass:A", %struct.TypeInfo** %typeInfoOrMeta_1
  call void @"kfun:A#<init>(kotlin.Int){}"(%struct.ObjHeader* %6, i32 6)
  br label %when_exit

when_exit:
  %7 = phi %struct.ObjHeader* [ %3, %when_body ], [ %6, %when_cond ]
  call void @"kfun:A#print(){}"(%struct.ObjHeader* %7)
  br label %return_block

return_block:
  %8 = phi i32 [ 0, %when_exit ]
  ret i32 %8
}
```

Результаты тестов

Так как текущая реализация SSA IR поддерживает не все конструкции языка Котлин, вместо готовых наборов тестов пришлось создавать свой, в котором некоторые идиоматические конструкции языка были заменены более примитивными. Тесты были разбиты на 4 категории.

1. Примитивные тесты, в которых нет аллокаций. Данные тесты позволяют оценить то, насколько улучшается работа LLVM при изменении способа порождения входного языка. В отличие от старого пайплайна в новом LLVM IR сразу порождается в SSA-форме, что упрощает работу оптимизатора (табл. 2).

2. Тесты, в которых аллоцированные объекты не *убегают*. Эта категория позволяет получить верхнюю оценку прироста от escape-анализа, так как при его использовании нагрузка на сборщик мусора падает практически до нуля.

3. Тесты, в которых аллоцированные объекты *убегают*. Цель данной категории – оценить эффективность межпроцедурной компоненты escape-анализа.

4. Сложные тесты, близкие к реальному продуктовому коду. В них большое количество виртуальных вызовов, и активно используется стандартная библиотека.

Тестовый стенд: MacBook Pro 2015.

Процессор: Intel i7-4980HQ.

Оперативная память: 16 GB DDR3.
Операционная система: macOS 10.15.3.

Таблица 2

Сравнение производительности

Table 2

Performance comparison

Режим компиляции	Среднее время работы тестов, с			
	1	2	3	4
Без SSA IR	33.9	51.6	68.4	107.3
С SSA IR	30.3	5.2	46.7	90.5
Отношение производительности	1.12	9.92	1.46	1.18

Заключение

В данной работе рассмотрены архитектура компилятора Kotlin / Native и ее основные проблемы. Представлено новое внутреннее представление, которое обладает свойствами, необходимыми для написания продвинутых оптимизаций. На его основе был реализован ряд анализов и оптимизаций, включая построение графа вызова функций и escape-анализ. Была реализована трансляция в данное представление из существующего древовидного, а также трансляция в LLVM IR. Удалось сохранить совместимость с текущим процессом компиляции, что обеспечивает возможность постепенного перехода на новое представление. Эксперименты показали, что данный подход может значительно увеличить производительность порождаемого кода.

Список литературы / References

1. **Lattner C., Adve V.** LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004, p. 75–86. DOI 10.1109/CGO.2004.1281665
2. **Bacon D. F., Cheng P., Rajan V. T.** A unified theory of garbage collection. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2004. DOI 10.1145/1035292.1028982
3. **Appel A. W.** SSA is functional programming. *ACM SIGPLAN Notices*, April 1998, vol. 33, no. 4, p. 17–20. DOI 10.1145/278283.278285
4. **Braun M., Buchwald S., Hack S., Leißa R., Mallon C., Zwinkau A.** Simple and Efficient Construction of Static Single Assignment Form. *Compiler Construction. Lecture Notes in Computer Science*, 2013, vol. 7791, p. 102–122. DOI 10.1007/978-3-642-37051-9_6
5. **Cytron R., Ferrante J., Rosen B. K., Wegman M. N., Zadeck F. K.** Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, October 1991, vol. 13, no. 4, p. 451–490. DOI 10.1145/115372.115320
6. **Bacon D. F, Sweeney P. F.** Fast static analysis of C++ virtual function calls. In: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. October 1996, p. 324–341. DOI 10.1145/236337.236371

7. **Choi J., Gupta M., Serrano M. J., Sreedhar V. C., Midkiff S. P.** Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis. *ACM Transactions on Programming Languages and Systems*, November 2003, vol. 25, no. 6, p. 876–910. DOI 10.1145/945885.945892

Материал поступил в редколлегию
Received
03.06.2020

Сведения об авторе

Боголепов Сергей Сергеевич, магистрант факультета информационных технологий Новосибирского государственного университета (Новосибирск, Россия)
s.bogolepov@g.nsu.ru

Information about the Author

Sergey S. Bogolepov, Master's Student, Faculty of Information Technologies, Novosibirsk State University (Novosibirsk, Russian Federation)
s.bogolepov@g.nsu.ru