Kotlin Performance

Kotlin
Оптимизация генерируемого байткода

Выполнил: Денис Жарков

Руководитель: Евгений Геращенко

Kotlin

- *Kotlin* — статически типизированный язык программирования, компилирующийся в JVM байт-код.
- Java-подобный лаконичный синтаксис
- High-order functions
- Приятные синтаксические конструкции

Постановка задачи

- *Бенчмарки*
  - Портирование известных бенчмарков
  - Сравнение результатов с аналогичным кодом на Java
- *Оптимизация*
  - Исправление найденных недостатков в генераторе байт-кода

Бенчмарки

- Большая часть известных бенчмарков на Java созданы для оценки качеств JVM
- Необходимо проверять качество генерации кода для различных синтаксических конструкций:
  - Простые операторы, которые есть и в Java, и в Kotlin: например *when* для целочисленных констант
  - Особенности Kotlin, которых нет в Java: *safe-call*, *elvis*, *inline*,..

Бенчмарки. JMH

JMH — фреймворк для написания микро-бенчмарков, запускаемых на JVM

- Позволяет избежать классических ошибок при написании бенчмарков
- Имеет множество методов измерения и настроек
- Интеграция с JVM

When

```
when (x) {
    null -> "null"
    is Int -> "Int"
    is String -> "String"
    else -> "Else"
}
```

Switch-like When

```
when (x) {
        1,2 -> "1,2"
        3,4 -> "3,4"
        5 -> "5"
        else -> "Else"
}
```

Switch-like When

- Каждый оператор switch в Java компилируется в одну из двух JVM-инструкций
  - *tableswitch* — работает за $O(1)$ времени и требует $O(max - min)$ памяти
  - *lookupswitch* — требует $O(n)$ памяти. По спецификации может работать за $O(n)$, но чаще всего реализован в виде двоичного поиска

Switch-like When

```
when (x) {
        1,2 -> "1,2"
        3,4 -> "3,4"
        5 -> "5"
        else -> "Else"
}
```

- В Kotlin оператор *when* компилировался в череду соответствующих условных переходов
- По этому поводу был реализован микробенчмарк, сравнивающий производительность *switch* в Java и оператора *when* с аналогичным набором значений в Kotlin
  - В случае “плотного” набора значений (*tableswitch*) реализация Kotlin проигрывает $\sim$ в 10 раз
  - В случае “разреженного” набора значений (*lookupswitch*) реализация Kotlin проигрывает $\sim$ в 2.5 раза

Switch-like When

- Было принято решение модифицировать алгоритм генерации байт-кода *when* в случае сравнения с целочисленными константами.
- Для выбора между *tableswitch*/*lookupswitch* используется эвристический алгоритм, подсмотренный в исходниках OpenJDK
- Изменения были приняты в основной репозиторий Kotlin

Inlining

```
(1..100).
    map { x -> x + 1 }.
    filter { x -> x % 2 == 0 }.
    sum()
```

Артефакты встраивания

- Избыточный боксинг
- “Лишние” переменные
- Избыточные GOTO-переходы

Избыточный боксинг

```
fun doAlmostNothing(y : Int, block : () -> Int) : Int {
    return block() + y
}

val x = one1
return doAlmostNothing(1) {
    x + one2
}
```

- Время работы бенчмарка примерно в два раза превосходит реализацию аналогичного кода (без встраивания)
- Возникла необходимость в поиске и анализе похожих проблем в существующем коде

Избыточный боксинг. Анализ

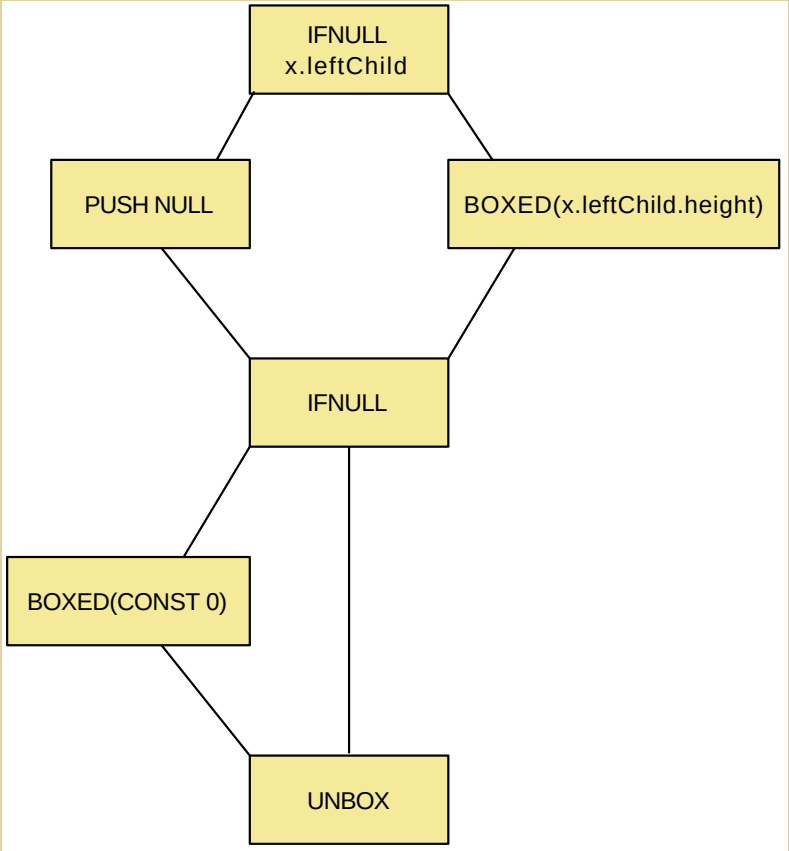
- *Control flow analysis* — классический подход к статическому анализу
- Был реализован анализатор, находящий в методах class-файлов участки байт-кода, где происходит избыточный боксинг
- Проанализированы уже существующие проекты на Kotlin (Kara, Kannotator, решения задач с Project Euler)
- Найдено 268 подозрительных мест

Избыточный боксинг. Оптимизация

- Реализована простая версия оптимизатора. Не обработаны случаи:
  - Упакованные значения, которые сохраняются в переменные
  - Упакованные значения из нескольких вершин графа потока исполнения
- Не сломан ни один из тестов компилятора
- Исправлено 60% подозрительных участков, найденных анализатором

Safe-call + Elvis

```
x.leftChild?.height ?: 0
obj?.field // Safe call
mayBeNull ?: 1 // Elvis operator
```



Результаты

- Разработан набор микробенчмарков, которые можно использовать в рамках CI для проверки отсутствия деградации кодогенератора
- Найдена и исправлена проблема неоптимальной генерации байт-кода оператора *when* для целочисленных констант
- Разработана утилита для обнаружения случаев избыточного боксинга значений и неоптимального кода, возникающего из-за совместного использования *Safe-call + Elvis*
- Реализован компонент, оптимизирующий некоторые случаи избыточного боксинга

Приобретенные навыки

- Базовое представление об устройстве JVM
- Азы написания бенчмарков для языков, компилируемых под JVM
- Основные понятия и методы статического анализа
- Плотная работа с байт-кодом (с помощью библиотеки Objectweb ASM)
- Первый опыт работы над настоящим языком

Спасибо за внимание!

Репозитории

- *https://github.com/bintree/kotlin*
- *https://github.com/bintree/kotlin-jmh-benchmarks*
- *https://github.com/bintree/kotlin-optimizer*