Конкурс исследовательских и проектных работ школьников

«Высший пилотаж»

Динамическая компиляция математических выражений с помощью Kotlin

Исследовательская работа

Направление «Computer science»

Автор: ***Постовалов Ярослав***, МБОУ Лицей № 130, г. Новосибирск,
11 класс физико-математического профиля

Научный руководитель: ***Нозик Александр Аркадьевич***,
кандидат физико-математических наук;
старший научный сотрудник МФТИ

Научные консультанты: ***Усманов Ильмир Ильдарович***,
ведущий программный инженер JetBrains GmbH;

Гринис Ролан,
Doctor of Philosophy, Mathematics,
научный сотрудник МФТИ,
технический директор GrinisRIT

Новосибирск 2022

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Одна из частых задач при разработке научного программного обеспечения (ПО) — это динамическая интерпретация математических выражений, то есть вычисление значения выражения, определенного либо в исходном коде, либо в строке, поступающей извне.

В общем, есть два подхода к решению этой задачи. Во-первых, это создание интерпретатора — при правильном выборе языка программирования, на котором написан интерпретатор, решение хорошо переносимо между разными платформами, но, как правило, приводит к меньшей производительности вычислений, сильно ограничивает ее. Во-вторых, это построение динамического компилятора [13] (компилятора just-in-time, JIT, компилятора «на лету») — это повышает верхнюю грань производительности ценою большей сложности реализации, т. к. в классическом виде для возможности работы динамического компилятора на нескольких процессорных архитектурах требуется реализовать генерацию машинного кода каждой из архитектур, более того, требуется совершить множество компромиссов между уровнем оптимизации и производительностью генерируемого кода.

Некоторые виртуальные машины прикладных языков программирования предоставляют конкурентоспособную производительность и механизм динамической загрузки программ в форме некоторого промежуточного представления. Среди них — широко известная виртуальная машина Java (Java Virtual Machine, JVM). Помимо Java, JVM поддерживает несколько других языков, включая современный и выразительный язык Kotlin. Так, например, Kotlin предоставляет среду исполнения скриптов, позволяющий компилировать и вычислять выражения «на лету». Однако, подобное использование компиляторов языков общего назначения сильно ограничено их размером — размер любой программы, использующей компилятор Kotlin, составил бы больше 47 МиБ[1], — а также сложностью применения и скоростью компиляции программ.

Цель данной работы — разработать универсальную относительно математических объектов, удобную и высокопроизводительную платформу динамического вычисления математических выражений. Этим требованиям не соответствует ни интерпретатор именно из-за невысокой предельной производительности, ни JIT-компилятор из-за описанных выше сложностей реализации.

Предложенная концепция реализована автором как часть библиотеки *KMath* [1, 2], библиотеки для Kotlin, реализующей вычисления как операции в алгебраических структурах над объектами Kotlin.

[1] Единица хранения информации мебибайт, один мебибайт равен 1 048 576 байтам.

Эта особенность дает некоторые преимущества при реализации и вычислении математических выражений. В частности, при построении компилятора была легко достигнута поддержка почти всех поддерживаемых библиотекой математическими объектами: целых чисел и чисел с плавающей точкой, чисел для длинной арифметики, комплексных чисел и кватернионов[2], многомерных массивов и т. д.

Важная особенность языка Kotlin, использованная KMath, — поддержка мультиплатформенного программирования, что позволило исследовать возможности динамической компиляции не только на JVM, но и на широко известной виртуальной машине языка JavaScript — V8 [23].

Для реализации динамической компиляции на основе существующей виртуальной машины необходимо было решить следующие задачи:

1. Разработать формальный язык или иные способы ввода выражения, неизвестного при компиляции приложения.
2. Разработать абстрактное синтаксическое дерево (Abstract Syntax Tree, AST) или иное промежуточное представление (Intermediate Representation, IR), описывающее этот язык.
3. И самое главное, сформулировать и реализовать методы компиляции этого IR, определив используемую виртуальную машину и способ загрузки кода в нее.

Прямых аналогов проекта, удовлетворяющих всем требованиям — универсальности трансляции выражений относительно типов, производительности того же порядка, что, например, Си, способности вычислять выражения, неизвестные на этапе компиляции — найдено не было. Но есть косвенные: требованиям частично соответствуют многие существующие технологии. Так:

1. Выражения, написанные на C++ достаточно быстры и за счет шаблонов могут быть верны для многих типов сразу, но почти неприменимы динамически.
2. Выражения на Python хорошо вычислимы динамически и общие для разных типов из-за типизации языка.
3. Выражения, компилируемые при помощи технологий JIT, таких как expr [22] высокопроизводительны, однако поддерживают очень немного типов данных.

[2] Кватернионы — система гиперкомплексных чисел, образующая векторное пространство размерностью четыре над полем вещественных чисел.

I. Динамическое вычисление выражений в библиотеке KMath

Пусть разработчику программы на Kotlin необходимо, чтобы программа могла вычислить некоторое математическое выражение, которое неизвестно на момент разработки программы, например, оно вводится в поля формы, или считывается из файла. В интерпретируемых языках типа Python это можно сделать с помощью функции eval(), которая может вычислить любое разрешенное синтаксисом языка выражение. Но в компилируемых языках, к которым можно отнести и Kotlin, такая функция не доступна.

Для реализации возможности динамической компиляции выражений в библиотеку KMath добавлены несколько дополнительных функций. Рассмотрим возможности библиотек KMath подробнее, т. к. особенности представления данных в ней являются критически важными для реализации динамического компилятора.

1. Обзор библиотеки KMath

Математические операции в KMath отделены от математических объектов. Пусть *алгебраический контекст* — объект, реализующий интерфейс Algebra<T> и содержащий методы — операции над типом T. Например, чтобы выполнить операцию «+», нужны два объекта некоторого типа T и соответствующая по типу реализация контекста, содержащего эту операцию, например Group<T>. У этой концепции следующие преимущества [1]:

1. Решаются проблемы с тем, что при бинарных операций чисел разных типов результирующий тип определяется по левому операнду, так что достигается коммутативность.
2. Одна и та же операция может быть реализована по-разному: так, KMath предоставляет и собственные хорошо переносимые реализации операций, и реализации на основе сторонних библиотекам, при этом они взаимозаменяемы.
3. Контекст (Algebra<T>, алгебраический контекст) может поддерживать контракты[3] над объектами с помощью проверок во время выполнения. Например, можно гарантировать единственную возможную форму многомерных массивов, верную при использовании контекста [1].

Основные интерфейсы контекстов имеют иерархию наследования, описанную на рисунке 1. Они более или менее соответствуют некоторым стандартным в математике алгебраическим структурам:

1. Group (группа) определяет операцию «+» и соответствующий нейтральный элемент0.
2. Ring (кольцо) добавляет операцию «•» и соответствующий нейтральный элемент1.

[3] Контрактов в значении предусловий относительно входных данных.

3. Field (поле) добавляет обратное по умножению, то есть «÷».

Некоторые контексты предоставляют дополнительные операций, такие как sin или exp.

Рассмотрим пример: в KMath представлена реализация Field<kotlin.Double> — DoubleField, реализующая необходимые операции с помощью стандартной арифметики с плавающей точкой двойной точности.

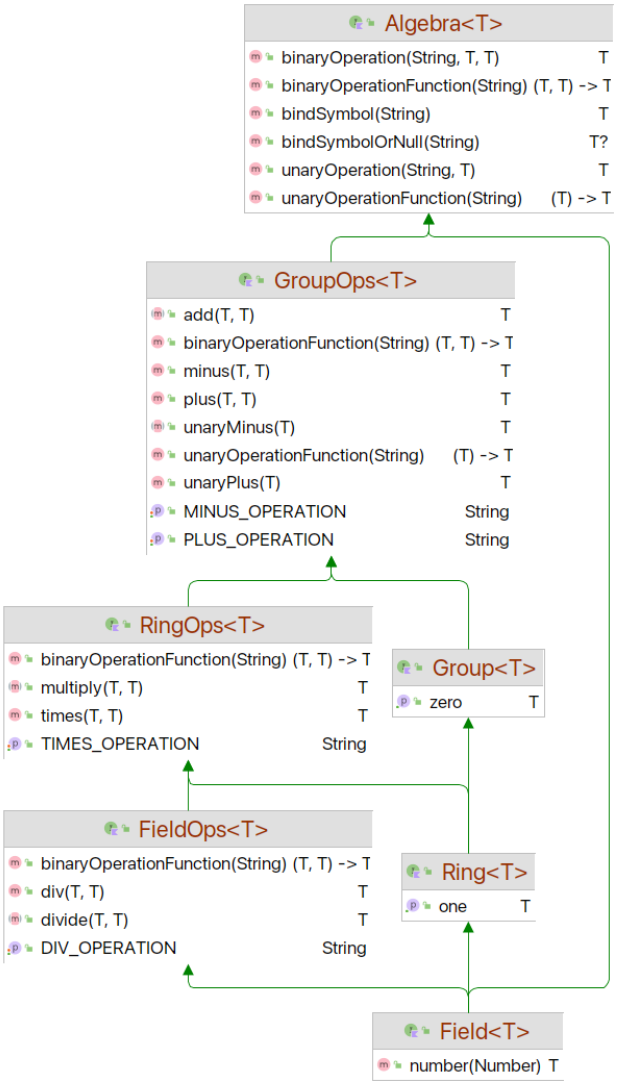

Рисунок 1. UML-диаграмма интерфейсов основных алгебраических контекстов

2. Программный интерфейс выражений в KMath

Элементы общей алгебры в архитектуре KMath дают некоторые преимущества для компиляции выражений: можно создать выражение, использующее только те операции, которые предоставлены некоторым интерфейсом из иерархии Algebra<T>, а затем использовать конкретный объект, соответствующий этому интерфейсу, чтобы вызывать операции и вычислять значение выражения. Однако, для того чтобы реализовать динамическую компиляцию, все еще требовались некоторые усилия.

Были внесены изменения в API KMath: в Algebra<T> были добавлены функции для динамической диспетчеризации операций. На момент написания работы ни один алгебраический контекст в KMath и известном коде, скомпонованным с ним, не объявляет тернарые операции, поэтому были добавлены только два метода: unaryOperation (для унарных операций) и binaryOperation (для бинарных операций). Эти методы принимают строку имени операции и значения аргументов, возвращая единственный результат. Также был добавлен метод bindSymbol, чтобы реализации контекстов могли разрешать значения констант, например i для описания алгебраических структур над комплексными числами. Позже были определены методы unaryOperationFunction и binaryOperationFunction, возвращающие объекты функциональных типов Kotlin: (T) -> T и (T, T) -> T — вместо мгновенного вычисления значений операций.

Следующая стадия реализации включила введение математических выражений как сущности в KMath: был добавлен интерфейс Expression, реализации которого должны определять метод invoke, принимающего значения параметров и вычисляющего значение (см. рис. 2). Была добавлена Algebra<Expression>, основанная на прямом вложении одних объектов Expression в другие, создающая называемые *функциональные выражения*, не представляющие интереса в работе, кроме как в сравнении производительности.

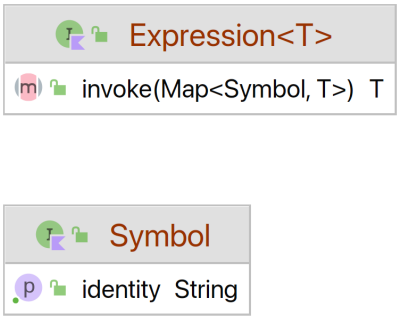

Рисунок 2. UML-диаграмма классов Expression и Symbol

3. Математическое синтаксическое дерево в KMath

MST (Mathematical Syntax Tree, математическое синтаксическое дерево) — это простое синтаксическое дерево, описывающее язык вычислений при помощи Algebra<T>, он состоит из следующей грамматики:

- terminal = symbol | number
- unaryEx = unaryOp, mst
- binaryEx = mst, binaryOp, mst
- mst = terminal | unaryEx | binaryEx

Начальный нетерминал — mst. Неопределенные грамматикой термы означают соответственно: symbol — символ, обозначающую некоторой константу, number — число-константу, unaryOp —

некоторый идентификатор унарной операции, binaryOp — некоторой идентификатор бинарной операции.

Данное синтаксическое дерево представлено в виде классов и интерфейсов в KMath с иерархией как на рисунке 3.

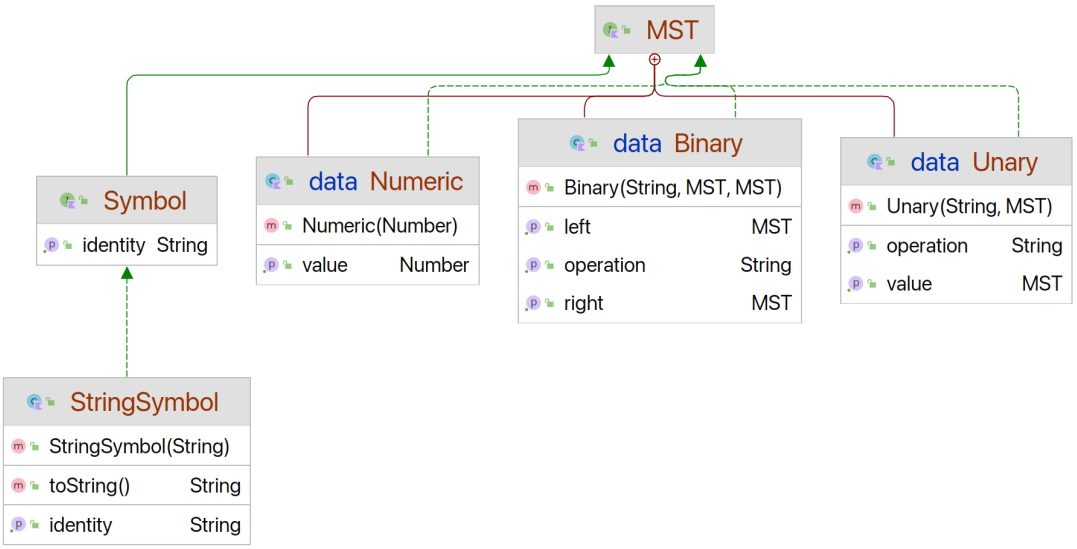

Рисунок 3. UML-диаграмма классов, описывающих MST

В KMath были добавлены три способа получить объект MST:

1. Выполнить синтаксический анализ некоторой строки с помощью более конкретной грамматики (см. файл ArithmeticsEvaluator.g4 в [2]), производящее некоторое множество объектов MST из строк вида sin(x)^2+25 (см. результат анализа этого выражения согласно этой грамматике на рисунке 4).
2. Явно инициализировать его с помощью конструкторов каждого из классов MST.
3. Применить алгебраический контекст над объектами MST, также добавленный в KMath.

MST может быть интерпретировано с помощью алгоритма обхода деревьев, однако, как выяснилось в ходе исследований производительности, это довольно медленный способ вычисления. Тем не менее, в API KMath встроен интерпретатор, потому что динамическая компиляция и загрузка программ недоступна в некоторых JVM, не поддерживается средой исполнения Kotlin/Native. Также он полезен при тестировании, отладке.

MST связан с API выражений с помощью класса MstExpression, являющегося парой из объекта MST и ссылки на алгебраическую структуру. Отличие между MstExpression и вызовом интерпретатора напрямую заключается в том, что в MstExpression листья MST типа symbol содержат не только константы, предоставляемые методом bindSymbol от объекта Algebra<T>, но и переменные выражения.

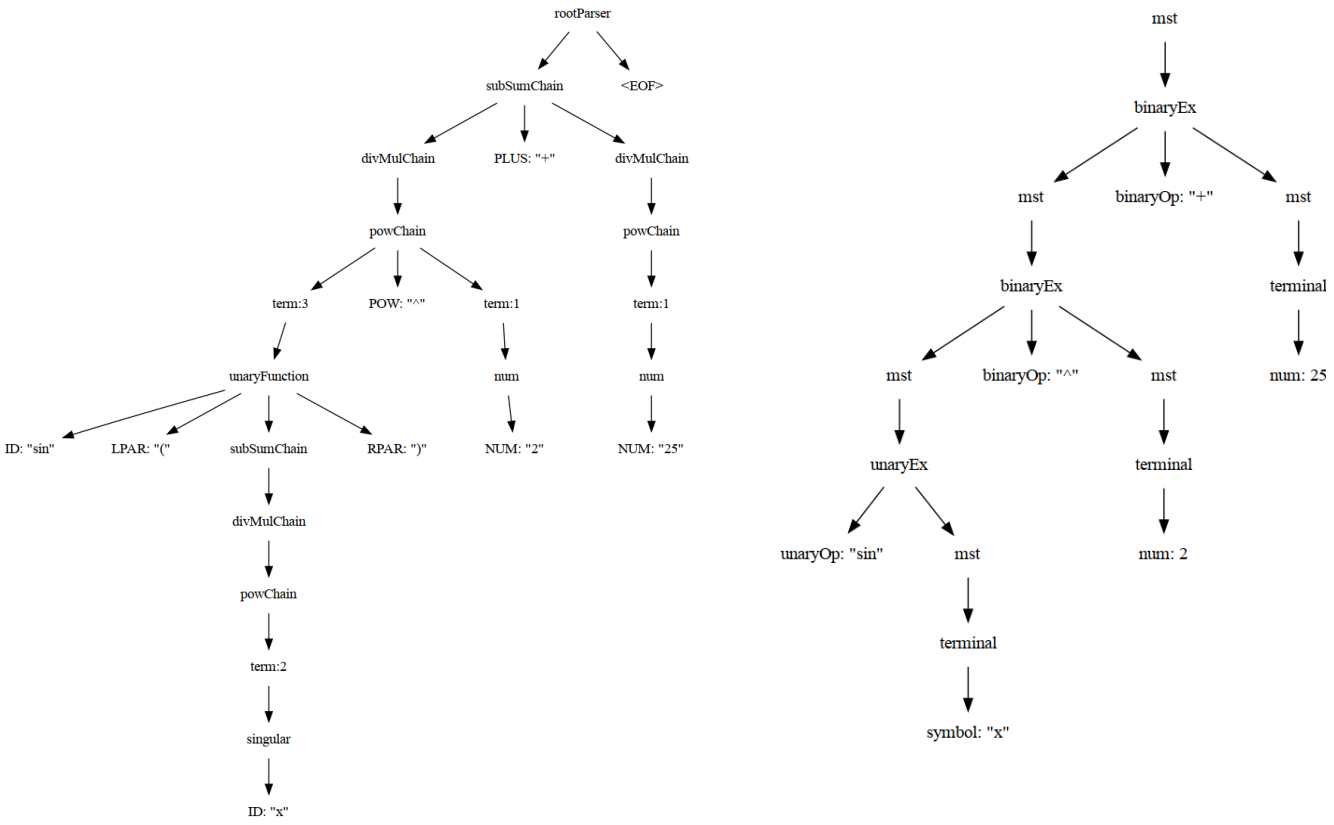

Рисунок 4. Анализа строки sin(x)^2+25 согласно ArithmeticsEvaluator и соответствующее MST

II. Компиляция математического синтаксического дерева

Основной задачей научно-исследовательской работы являлся выбор наиболее быстрого способа динамической компиляции выражений, представленных в виде MST.

Были рассмотрены четыре метода трансляции MST, два из которых оказались достаточно производительными и универсальными — способными использовать любые алгебраические контексты: как встроенные в KMath, так и объявленные пользователем. Данные тестирования, выполненного для каждого из выражений приведены в приложении 1.

Рассмотрим реализованные методы динамической компиляции подробнее.

1. Генерация классов JVM

Идея компиляции MST в байт-код Java — это динамически сгенерировать и загрузить классы Java по данному экземпляру MST. При этом сгенерированный класс должен реализовывать интерфейс Expression с правильными сигнатурами типов, работать равносильно интерпретатору и делегировать все вызовы операций к некоторому алгебраическому контексту KMath — чтобы быть универсальным.

Библиотека *ObjectWeb ASM* [4] была применена для манипуляции байт-кодом, т. к. это наиболее легковесная и широко применяемой в разработке компиляторов промышленных языков программирования для JVM — в том числе самого языка Java.

При реализации данного метода были обнаружены две значительные проблемы.

Во-первых, это боксинг. Приведения типов боксинг и анбоксинг [12] («упаковка» и «распаковка») часто выполняются на JVM из-за механизма стирания типов, появившегося для поддержки обобщенных типов, и они часто снижают производительность вычислений. Expression<T> — обобщенный интерфейс, поэтому для аргументов и результатов должно применяться преобразование боксинга. Тем не менее, на уровне виртуальной машины преобразование боксинга может оптимизироваться при помощи escape-анализа и скаляризации, поэтому производительность сгенерированных выражений была также исследована на альтернативной реализации JVM *GraalVM* [6] (см. подсекцию 5).

Следующая проблема была в том, каким образом составлять инструкции вызова фактического метода JVM, реализующего те или иные алгебраические операции. Были рассмотрены четыре решения:

1. Выполнение поиска нужного метода в классе объекта Algebra<T> при помощи рефлексии Java, в сгенерированное выражение вставляется прямой вызов этого метода.
2. Вместо прямого вызова, каждый раз вызываются новые функции Algebra<T>, описанные ранее, unaryOperation и binaryOperation.
3. Используются прямые вызовы, но только если пользователь предоставляет таблицу, сопоставляющую идентификаторы операций Algebra<T> и сигнатуры методов в объекте.
4. На этапе компиляции MST от объекта Algebra<T> единожды применяются функции unaryOperationFunction и binaryOperationFunction, а сгенерированные выражения содержат ссылки на полученные объекты функций.

У каждой из опций есть преимущества и недостатки, описанные в таблице 1.

Таблица 1

	1. Поиск рефлексией	**2. Прямые динамические вызовы**	**3. Вызов метода по таблице**	**4. Непрямые динамические выводы**
Проблема боксинга	Только для возвращаемого значения	И для аргументов, и для значения	При наличии таблицы — только для возвращаемого значения	И для аргументов, и для значения — но вероятны оптимизации

Не работает, если имя операции не соответствует имени метода	Да	Нет	Нет	Нет
Требует дополнительные параметры компилятора	Нет	Нет	Да	Нет
Сопоставляет имя операции с методом	Только если метод не найден	При каждом вызове	Нет	Только при компиляции

При разработке модуля реализованы были два из этих способов. Первый — поиск при помощи рефлексии — работал некорректно в случаях, если имя операции не соответствовало методу в классе алгебраического контекста или семантика операции не соответствовала семантике метода. Реализация алгоритма компиляции также была запутанной из-за необходимости правильных приведений типов вызываемых методов и активного применения рефлексии.

Во второй попытке был применен четвертый вариант — объекты функциональных типов агрегировались с помощью методов unaryOperationFunction, binaryOperationFunction. Этот алгоритм привел к большим потерям производительности из-за боксинга, но оказался более стабильным и универсальным.

```
import java.util.*;

import scientifik.kmath.asm.internal.*;
import scientifik.kmath.expressions.*;
import scientifik.kmath.operations.*;

public final class AsmCompiledExpression_1073786867_0 implements Expression<Double> {
    private final RealField algebra;

    public AsmCompiledExpression_1073786867_0(RealField algebra) {
        this.algebra = algebra;
    }

    public final Double invoke(Map<String, ? extends Double> arguments) {
        return (Double) algebra
                .add(((Double) MapIntrinsics.getOrFail(arguments, "x")).doubleValue(), 2.0D);
    }
}
```

Листинг 1. Байт-код при компиляции выражения x+2 вариантом 1 (декомпилированный в Java

```java
import java.util.Map;
import kotlin.jvm.functions.Function2;
import space.kscience.kmath.asm.internal.MapIntrinsics;
import space.kscience.kmath.expressions.Expression;
import space.kscience.kmath.expressions.Symbol;

public final class CompiledExpression_1073786867_0 implements Expression<Double> {
    private final Object[] constants;

    public final Double invoke(Map<Symbol, ? extends Double> arguments) {
        Double var2 = (Double)MapIntrinsics.getOrFail(arguments, "x");
        return (Double)((Function2)this.constants[0]).invoke(var2, (Double)this.constants[1]);
    }

    public CompiledExpression_1073786867_0(Object[] constants) { this.constants = constants; }
}
```

Листинг 2. Байт-код при компиляции выражения x+2 вариантом 4 (декомпилированный в Java)

Сравним подробно байт-код, сгенерированный разными алгоритмами генерации в декомпилированной в Java форме: варианта 1 — листинг 1, варианта 4 — листинг 2.

Оба класса сгенерированы из выражения x+2 в контексте DoubleField (RealField в старых версиях KMath), реализующем, ExtendedField<kotlin.Double>, так что оба класса реализуют Expression<kotlin.Double>. Флаги доступа, способ именования классов, объявленные методы и сигнатуры типов (за исключениям типа ключа карты аргументов, в актуальной версии он заменен на Symbol) одинаковы. Первое отличие — поля классов. Первый вариант генератора создавал два поля: ссылку на объект Algebra<T> и массив констант типа java.lang.Object[], которые нельзя разместить в пул констант класса, как, например, числа или строки. Итоговый генератор создает только одно поле — констант, хранящее и собственно константы, и объекты функциональных типов, создаваемые методы Algebra<T> unaryOperationFunction и binaryOperationFunction. Еще одно изменение — значения аргументов кэшируются в локальные переменные, что приводит к лучшей производительности на больших выражениях. Конструкторы выражений также отличаются, чтобы соответствовать создаваемым полям. Вне зависимости от способа генерации, после загрузки экземпляры классов создаются при помощи рефлексии.

После исследования производительности (подсекция 5) выяснилось, что в итоге боксинг приводит к высокой потере производительности для примитивных типов по сравнению со статически скомпилированными выражениями, так что был добавлен дополнительный метод компиляции, *специализированный* для контекстов IntRing, LongRing и DoubleField. Результирующие выражения обращаются напрямую к арифметике и математическим функциям JVM, как на листинге 3. Более того, чтобы избежать потери производительности, были добавлены дополнительные интерфейсы IntExpression, LongExpression и DoubleExpression, которые при

помощи механизма SymbolIndexer (объект, который для данного значения Symbol, получает некий числовой индекс, которым можно пользоваться на протяжении всего жизненного цикла выражения) способен принимать в качестве аргументов обычный массив чисел, что позволило для данных весьма часто используемых типов избежать всяких потерь производительности.

```
import java.util.Map;
import space.kscience.kmath.asm.internal.MapIntrinsics;
import space.kscience.kmath.expressions.DoubleExpression;
import space.kscience.kmath.expressions.Symbol;
import space.kscience.kmath.expressions.SymbolIndexer;

public final class CompiledExpression_3530400_0 implements DoubleExpression {
    private final SymbolIndexer indexer;

    public SymbolIndexer getIndexer() { return this.indexer; }

    public double invoke(double[] arguments) {
        double var2 = arguments[0];
        return Math.sin(var2);
    }

    @NotNull public final Double invoke(Map<Symbol, ? extends Double> arguments) {
        double var2 = ((Double)MapIntrinsics.getOrFail( $this$getOrFail: arguments,  key: "x")).doubleValue();
        return Math.sin(var2);
    }
}
```

Листинг 3. Байт-код при компиляции выражения sin(x) методом специализации. Виден прямой вызов java.lang.Math.sin (декомпилированный в Java)

2. Генерация исходного кода на JavaScript

Применив *Kotlin Multiplatform* к созданному модулю библиотеки, легко удалось перенести MST на *Kotlin/JS* и *Kotlin/Native*. При этом на каждой из платформ оказалось возможным отдельно реализовать функциональность, подобная динамической компиляции на JVM.

После поддержки JVM разработка компилятора на основе Kotlin/JS была прямолинейна. Идея хранения функций вместо ссылки на объект Algebra<T> была выведена из бэкенда для JVM, то есть сгенерированная функция принимает массив констант, содержащих и ссылки на функции KMath, и просто значения — константы выражения, и массив аргументов.

Что касается использованного инструментария, то была применена спецификация *ESTree* [9] для описания классов AST JavaScript (JS), а пакет *astring* [8] в качестве фреймворка генерации кода по созданному синтаксическому дерева. В ходе реализации был сделан между созданием исходного кода путем ручной конкатенации его фрагментов и созданием объекта AST и последующей генераций кода по нему — в пользу последнего из-за простоты реализации.

Компилятор из MST в JS генерирует функцию, а затем оборачивает ее в объект Expression из KMath. Пример подобной функции приведен в листинге 4.

```
var executable = function (constants, arguments) {
    return constants[1](constants[0](arguments, "x"), 2);
};
```

Листинг 4. Пример сгенерированной функции JS

3. Генерация WebAssembly IR

WebAssembly [15] (также известен как WASM) — открытый стандарт переносимого IR программ, и в этом исследовании была сделана попытка генерировать код WebAssembly; однако, компилятор в байт-код WASM оказался в итоге более ограниченным, чем в байт-код JVM или исходный код JS.

Чтобы достичь динамической компиляции, сам транслятор MST был написан при помощи Kotlin/JS. У полученной реализации есть следующие особенности:

1. Из-за медленной интероперабельности между JS и WASM, встроенные математические функции Kotlin/JS, которые на самом деле просто делегируют все вычисления к объекту Math, были недоступны, не говоря уже о возможности работы с функциями из контекстов KMath.
2. Поддерживаются только два типа WASM: f64 и i32 — контексты KMath DoubleField и IntRing соответственно. i64 недоступен без применения экспериментальных возможностей V8, обеспечивающих связь между типом WASM i64 и типом JS bigint. f32 недоступен по аналогичной причине — JavaScript попросту не предоставляет типа, отображающего числа с плавающей точкой одинарной точности.
3. Элементарные функции над f64, такие как синус и косинус, необходимые для поддержки операций из DoubleField, были взяты из библиотеки libm (так же известной как math.h [11]), предварительно скомпилированной под WASM и частично добавленной к изначальному состоянию модуля WASM. Вся прочая арифметика f64 доступна в WASM на уровне встроенных операций.

```
(func $executable (param $0 f64) (result f64)
  (f64.add
    (local.get $0)
    (f64.const 2)
  )
)
```

Листинг 5. Пример результирующего WASM IR в форме WAT

Описанный бэкэнд использует библиотеку binaryen [10], чтобы упростить генерацию IR и выполнять некоторые оптимизации.

4. Генерация LLVM IR

LLVM [14] (Low-level Virtual Machine) — проект инфраструктуры для создания компиляторов и сопутствующих им утилит, разработанный вокруг IR, предоставляющего собой переносимый и высокоуровневый язык ассемблера. LLVM используется как бэкэнд компилятора Kotlin/Native и был рассмотрен в качестве возможной цели компиляции. Однако, эта возможность была отвергнута в основном по двум причинам:

1. Невозможна универсальность компиляции относительно типа выражений из-за большой сложности интероперабельности сгенерированного кода с Kotlin/Native.
2. Монолитная архитектура LLVM и ее низкая производительность, особенно на высоких уровнях оптимизации, делают ее неприменимой для динамической компиляции, как минимум для примитивных вычислений. Упомянутые ранее представления имеют легкую или встроенную в саму платформу инфраструктуру для генерации кода во время выполнения.

5. Исследование производительности

Были проведены бенчмарки новых API вычисления выражений на компьютере с центральным процессором: Intel Core i5 6400 (тактовая частота — 3,196 ГГц, микроархитектура — Skylake), объемом ОЗУ: 15,977 ГиБ[4] и операционной системой: Kubuntu 21.10 Impish Indri.

Исследованные реализации API использовали вычисляли значение следующей формулы один миллион раз с использованием стандартной арифметики над числами с плавающей точкой двойной точности:

$$2x + \frac{2}{x} - \frac{16}{\sin x} \quad (1).$$

Для исследования проивзодительности на JVM использовался *JMH* [5] (Java Microbenchmark Harness), поставляемый в составе инструмента kotlinx-benchmark [7], в режиме пропускной способности с 3 итерациями и одной итерацией разогрева при времени 5 с на итерацию. Измерения на JVM представлены в приложении 2. Время компиляции не учитывалось.

Измерения, проведенные на JavaScript представлены в приложении 3. Использовалось измерение времени единичного запуска. Время компиляции также не учитывалось.

Из данных бенчмарков можно сделать следующие выводы:

[4] Единица хранения информации гибибайт, один гибибайт равен 1 073 741 824 байтам.

1. GraalVM быстрее Hotspot для вычисления выражений, скорее всего, из-за упомянутой в подсекции 1 лучшей скаляризации, из-за которой снизились накладные расходы на боксинг.
2. Есть пока что не исследованное падение производительности интерпретации MST на JS.
3. Статические вычисления значительно быстрее динамических из-за высоких накладных расходов последних, связанных согласно данным профилирования запусков JVM с использованием java.util.HashMap для передачи значений аргументов. Тем не менее, их удалось частично избежать путем разрешения передачи массивов аргументов при использовании типов Int, Long, Double.
4. Измерения, проведенные на JVM, необходимо будет провести повторно на будущих реализациях JVM, включающих результаты проекта *Valhalla* [17], включающего в себя JDK Enhancement Proposals (JEP) 218 [18], так что проблема боксинга, вызванная обобщенными типами, будет полностью решена на уровне JVM.

ЗАКЛЮЧЕНИЕ

Исследование показывает потенциал динамически собранных выражений даже в критических местах. На JVM динамически скомпилированные выражения сравнимы, имеют тот же порядок скорости вычисления со статически скомпилированными. На выражениях, работающих в примитивных типах (int, long, double) благодаря специализации и передаче аргументов через массив производительность повышена до той же, что и у статических выражений.

Динамическая компиляция может быть полезна в области численных стохастических методов, то есть чтобы моделировать распределения, введенные динамически пользователями некоторой системы [21].

Структура MST была побочным продуктом исследования, но оказалась полезной сама по себе. Во-первых, MST может использоваться для несложных символьных вычислений. Во-вторых, реализована поддержка автоматического дифференцирование MST на основе библиотек Kotlin∇ [16] и Symja [19], при этом динамическое компилирование повышает эффективность выражений-производных. В-третьих, KMath уже поддерживает визуализацию объектов MST как в LaTeX, так и MathML: это применяется для интеграции KMath в Jupyter Notebook (см. рис. 5).

Автор хотел бы поблагодарить своего научного руководителя Александра Нозика, консультантов Ролана Гриниса и Ильмира Усманова, соавторов всех прочих не связанных с разработанными или улучшенными в рамках работы модулей библиотеки KMath (Андрея Кислицына, Евгения Желенского, Александру Новикову и др.), а также Breandan Considine за обсуждение и ревизию английской версии работы. Проект KMath разрабатывается в кооперации между МФТИ и JetBrains Research.

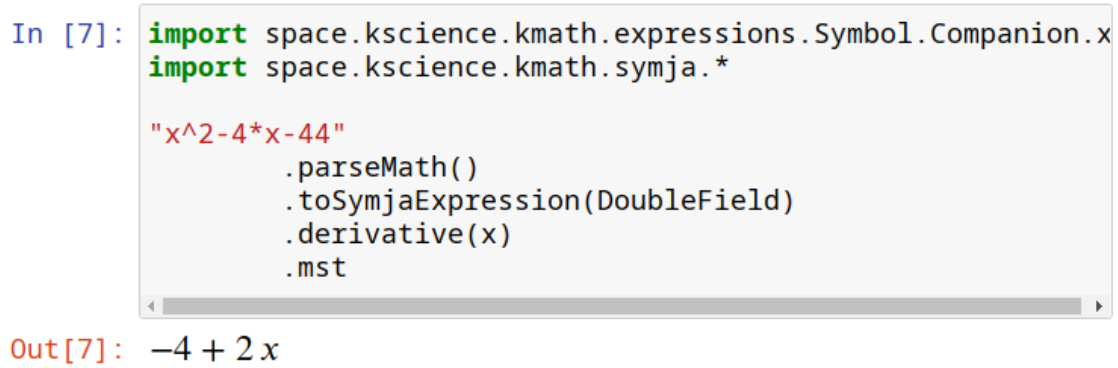

Рисунок 5. Пример дифференцирования и визуализации объекта MST

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Nozik, A. Kotlin language for science and Kmath library / A. Nozik // AIP Conference Proceedings. — 2019. — Окт. — Т. 2163, № 1. — С. 040004. — doi:10.1063/1.5130103
2. mipt-npm/kmath: 0.3.0-dev-8 / I. Postovalov [и др.]. — Вер. v0.3.0-dev-8. — 05.2021. — doi:10.5281/zenodo.4743708
3. JetBrains. Kotlin Programming Language [Электронный ресурс] / JetBrains. — Вер. 1.6.0-RC. — 12.10.2021. — URL: https://kotlinlang.org (дата обр. 28.10.2021).
4. ObjectWeb ASM [Электронный ресурс] / E. Bruneton [и др.]. — Вер. 9.1. — 06.02.2021. — URL: https://asm.ow2.io (дата обр. 28.10.2021).
5. jmh [Электронный ресурс] / A. Shipilev [и др.]. — Вер. 1.21. — 04.05.2018. — URL: https://openjdk.java.net/projects/code-tools/jmh/ (дата обр. 28.10.2021).
6. Oracle Corporation. GraalVM [Электронный ресурс] / Oracle Corporation. — URL: https://www.graalvm.org (дата обр. 28.10.2021).
7. kotlinx-benchmark [Электронный ресурс] / I. Ryzhenkov [и др.]. — Вер. 0.3.1. — 01.05.2021. — URL: https://github.com/Kotlin/kotlinx-benchmark (дата обр. 28.10.2021).
8. Astring [Электронный ресурс] / D. Bonnet [и др.]. — Вер. 1.7.0. — 07.02.2021. — URL: https://github.com/davidbonnet/astring (дата обр. 28.10.2021).
9. ESTree [Электронный ресурс]. — URL: https://github.com/estree/estree (дата обр. 13.02.2021).
10. Binaryen [Электронный ресурс]. — Вер. 100.0. — 03.03.2021. — URL: https://github.com/WebAssembly/binaryen (дата обр. 28.10.2021).
11. ISO. ISO/IEC 9899:2011 Information technology — Programming languages — C / ISO. — Geneva, Switzerland : International Organization for Standardization, 12.2011. — Гл. 7.12. 683 (est.) — URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853 (дата обр. 28.10.2021).
12. The Java® Language Specification, Java SE 11 Edition [Электронный ресурс] / J. Gosling [и др.]. — Oracle Corporation, 09.2018. — URL: https://docs.oracle.com/javase/specs/jls/se11/jls11.pdf (дата обр. 13.02.2021).
13. Languages, Compilers, and Runtime Systems [Электронный ресурс]. — URL: https://www.eecs.umich.edu/eecs/research/area.html?areaname=languages-compilers (дата обр. 15.02.2021).

14. Lattner, C. LLVM: a compilation framework for lifelong program analysis transformation / C. Lattner, V. Adve // International Symposium on Code Generation and Optimization, 2004. CGO 2004. — 2004. — С. 75—86. — doi:10.1109/CGO.2004.1281665
15. Bringing the web up to speed with WebAssembly / A. Haas [и др.] //. — 06.2017. — С. 185—200. — doi:10.1145/3062341.3062363
16. Considine, B. Kotlin∇: A shape-safe DSL for differentiable programming / B. Considine, M. Famelis, L. Paull //. — 2019.
17. Valhalla [Электронный ресурс]. — URL: https://openjdk.java.net/projects/valhalla/ (дата обр. 22.02.2021).
18. JEP 218: Generics over Primitive Types [Электронный ресурс]. — URL: https://openjdk.java.net/jeps/218 (дата обр. 22.02.2021).
19. Symja Library [Электронный ресурс] / A. Kramer [и др.]. — Вер. 2.0.0. — URL: https://github.com/axkr/symja_android_library (дата обр. 01.11.2021).
20. Wolfram Language [Электронный ресурс]. — URL: https://www.wolfram.com/language/ (дата обр. 05.11.2021).
21. Постовалов Я. С., Васильев Т. В., Черкашин Д. А. Проект компьютерной системы для выбора и исследования моделируемых вероятностных распределений: сб. ст. Математика : Материалы 58-й Междунар. науч. студ. конф. 10—13 апреля 2020 г. / Новосиб. гос. ун-т. — Новосибирск : ИПЦ НГУ, 2020. — 186 с.
22. expr [Электронный ресурс] / S. Zaitsev [и др.]. — URL: https://github.com/zserge/expr (дата обр. 10.11.2021).
23. V8 JavaScript Engine [Электронный ресурс]. — URL: https://v8.dev/ (дата обр. 11.21.2021).

Приложение №1: Примеры — тестовые ситуации методов компиляции выражений

Выражение	Аргументы	Результат
x	x = 1	1
+x	x = 2.0	2.0
-x	x = 2.0	-2.0
x + x	x = 2.0	4.0
sin(x)	x = 0.0	0.0
cos(x)	x = 1.0	0.0
x - x	x = 2.0	0.0
x / x	x = 2.0	1.0
x^2	x = 2.0	4.0
(+((x-((1+1)*2+1+2))*3-1)+1)*2	x = 3	-24
+(1*2+1+1)+((-(x+((1+1)*2+1))+3)*3-1+1+(1*0.5+2*1))+0	x = 2.0	-5.5
x + y	x = 1, y = 1	2

Приложение №2: Измерения производительности на JVM

Использованные JVM**:**

1. OpenJDK Hotspot (build 11.0.13+8-LTS),
2. OpenJDK GraalVM CE 21.3.0 (build 11.0.13+7-jvmci-21.3-b05).

Кодовое имя измерения	**Описание**	**Средняя пропускная способность (больше — лучше)**	
		1. Hotspot	**2. GraalVM**
functional	Функциональное выражение	3,656 Гц	5,137 Гц
mst	Интерпретирование MST	2,136 Гц	3,946 Гц
asmGeneric	Компиляция при помощи ObjectWeb ASM без специализации	6,449 Гц	10,741 Гц
asmPrimitive	ASM со специализацией	10,140 Гц	21,115 Гц
asmPrimitiveArray	ASM со специализацией и передачей аргументов через массив	34,357 Гц	25,407 Гц
raw	Статическая реализация Expression	15,731 Гц	20,578 Гц
justCalculate	Статическое вычисление без Expression	35,811 Гц	27,436 Гц

Замечания:

- Единица измерения Герц в измерениях используется в как единица частоты выполнения операций, которую также измеряют в операциях в секунду.
- Бенчмарк raw включает накладные расходы от применения интерфейса Expression (боксинг результирующего значения, хранение аргументов в хэш-таблице) для правильного сравнения с прочими его результатами, в то время как бенчмарк justCalculate выполняет вычисление на используемой виртуальной машине напрямую.
- Бенчмарки asmPrimitive и asmPrimitveArray отличаются тем, что в первом аргументы передаются через объект HashMap, а в последнем — через массив.

Приложение №3: Измерения производительности на JavaScript

Использованные среды исполнения JS**:**

1. Node.js 14.15.4 (V8 8.4.371.19-node.17),
2. Node.js 14.16.1 (V8 8.4.371.19-node.18) в составе GraalVM.

Кодовое имя измерения	**Описание**	**Время единого запуска (меньше — лучше)**	
		1. Node.js	**2. GraalVM Node.js**
functional	Функциональное выражение	2,42 с	10,4 с
mst	Интерпретирование MST	18 с	25,2 с
wasm	Компиляция в WASM	1,46 с	5,06 с
estree	Компиляция в функцию JS	1,53 с	3,72 с
raw	Статическая реализация Expression	1,16 с	3,23 с
justCalculate	Статическое вычисление без Expression	0,031 с	0,104 с