AST-based compilation of mathematical expressions in Kotlin

Iaroslav Postovalov
JetBrains Research (Nuclear Physics Methods group)
Novosibirsk, Russia
ORCID: 0000-0002-7221-365X
postovalovya@gmail.com

***Abstract*—Interpreting mathematical expressions at runtime is a standard task in scientific software engineering fields. There are different approaches to this problem from creating an embedded domain specific language (eDSL) with its own parser and interpreter specifically just for that task, to using a full-fledged embedded compiler. This article is dedicated to a middle-ground solution implemented in the KMath library. KMath uses the Kotlin object builder DSL and its own algebraic abstractions to generate an AST for mathematical operations. The AST is then compiled just-in-time to generate JVM bytecode generation. A similar approach is tested on other Kotlin platforms.**

***Index Terms*—Dynamic compiler, Software libraries, Software performance**

I. Introduction

The dynamic interpretation of mathematical expressions is a common goal in many scientific programming settings. Usually, an expression is defined as source code or an external string which must be evaluated within the program context (possibly limited by security considerations).

In general, there are two approaches to dynamic execution: pure interpretation is more portable but typically slower, and dynamic compilation (i.e. JIT [1], or just-in-time compilation) can be faster but is more complex. As the goal of this research is to provide a universal and high-performance dynamic execution engine, pure interpretation does not meet our requirements, and implementing a complete JIT compilation cycle is too complex, requiring many tradeoffs between the level of optimization and the performance of compilation.

Various runtimes offer competitive performance and a mechanism to load the intermediate representation (IR) dynamically, among them, the well-known *Java Virtual Machine* (JVM). The JVM supports a variety of language frontends, including a modern, expressive language called Kotlin [2].

A reasonable question to ask is why one would implement custom compiler infrastructure instead of using a existing solution? For example, Kotlin provides a scripting engine for compiling expressions on-the-fly. However such general-purpose compilers are limited by their size and complexity (e.g., `kotlin-compiler-embeddable` is around 40 MiB), and the compilation performance is usually slow.

The alternative described in this paper is implemented in a library called *KMath* [3], [4], a Kotlin-based mathematical library, which implements a generic set of elements and algebraic operations over them using abstract algebra.

The approach taken by KMath offers a number of benefits for defining and evaluating mathematical expressions. In particular, users can easily implement expressions on double-precision floating-point numbers and define custom operators on all supported algebraic structures.

II. KMath Principles

Mathematical operations in KMath are generally separated from mathematical objects. To perform an operation, say $+$, one needs two objects of type `T` and an algebra context, which delegates to a generic object, say `Space<T>`. Separating operations from objects has several advantages:

- Multiple operations may be valid depending on the application. For geometric applications, a matrix has an elementwise sum but no corresponding operator for multiplication. While NumPy semantics allow elementwise addition, multiplication and other operators, mixing these operations in a single type can lead to confusion and complicate implementation.
- Multiple implementations of the same operation may be possible. For example, *KMath* supports binding to external libraries that may be used interchangeably.
- The context (called `Algebra` or algebraic context) could store information required to provide additional runtime guarantees. For example, it is possible to guarantee only a specific shape of n-dimensional arrays are valid in a given context or lexical scope. [5]

Mathematical contexts have the following hierarchy:
`Field <: Ring <: Space <: Algebra`
These interfaces loosely fulfill the standard mathematical definition following their name:

1) Space defines addition, an additive identity (i.e., 0), and scalar multiplication.
2) Ring adds multiplication and a multiplicative identity (i.e., 1).
3) Field define an inverse operation of multiplication, i.e. division.

A typical implementation of `Field<T>` is the `RealField` which works on doubles, and `VectorSpace` for `Space<T>`. In some cases, the algebra context can hold additional operations such as exp or sin, which are inherited from the appropriate interface.

III. Design of Expressions API

KMath abstract algebra offers additional benefits for expression compilation. One can create a generic expression that uses only operations, provided by an `Algebra` contract and then use a specific `Algebra` instance to perform operations and compute the value of the expression. Still, it requires some additional work to make it convenient for dynamic expression compilation.

The first major change of KMath core API is the addition of dynamic operation dispatching to the primary marker interface `Algebra<T>`. Currently, no algebraic context of KMath declares ternary operations, so only `unaryOperation` and `binaryOperation` methods were added to call an operation dynamically by its name and operands. Also, the `symbol` method has been added so that `Algebra<T>` could declare constants like the imaginary i in the context of complex numbers. Also, `unaryOperationFunction` and `binaryOperationFunction` companion functions have been added with the only difference that they return the Kotlin function type object instead of the value of the operation.

The second stage of implementation was about submitting expressions as an entity in KMath.

In the current API, instances of `Expression` should declare a function `invoke` which takes bindings and evaluates the expression.

The most basic implementations of `Expression` are the so-called *functional expressions*, which are organized as a tree of `Expression` objects.

IV. The MST Structure

MST (Mathematical Syntax Tree) is a primitive abstract syntax tree, and it describes a certain set of expressions with only four kinds of nodes:

- terminal numeric node (e.g., 42),
- terminal symbolic node (e.g., i),
- unary operation node (e.g., $\sin(\text{arg})$),
- binary operation node (e.g., $\text{left} + \text{right}$).

There are three ways to obtain MST instances:

1) Parse a string using a grammar that can produce a certain set of MST nodes.
2) Construct it directly.
3) Use a special KMath context where all the operations create an MST.

The MST can be interpreted by simple recursive traversal, and it is actually the slowest way of expression execution, even slower than functional expressions (as will be seen from the benchmarks' data), and KMath API provides such an interpreter for three reasons: dynamic compilation is restricted in several VM environments, the dynamic compilation is not implemented for Kotlin/Native, and the interpreter is useful for testing reasons.

`MST` is connected to `Expression` API with `MstExpression` class, which is basically a pair of an MST node, and an algebraic structure reference. The only difference between `MstExpression` and direct `MST` interpretation is that in `Expression` implementation symbolic nodes are loaded not only with the constants and literals of the target algebraic context, but also with the expression symbols too.

Four other ways were considered of `MST` translation. Two turned out to be performance successful and universal in the sense that they can compile `MST` instances for any algebraic structure—both user-declared and those loaded from any KMath module.

V. Java Class Generation

The goal of `MST` compilation to Java bytecode is to get and load Java class dynamically from an `MST` instance. The generated class should implement the `Expression` interface with valid type parameters, be consistent with the interpreter and delegate all the operations directly to a KMath algebraic structure to be universal.

ObjectWeb ASM [6] was picked as a bytecode manipulation framework since it is considered to be the most lightweight and is used by many industrial strength languages' compilers.

There were two major problems.

The first one is boxing. Boxing and unboxing type conversions [7] are often performed in JVM because of type erasure and degrade calculation performance, and since `Expression` interface is a generic one, boxing conversions are performed there, so the number of these conversions should be minimized. It is possible to optimize out boxing with escape analysis and scalar replacement, but the performance of generated expressions on alternative JVM implementations like GraalVM [8] has to be investigated.

The second one is the way of acquiring a method signature that will call the needed algebraic operation.

Four options for solving these problems have been considered:

1) The method to invoke is searched within the given `Algebra` object with Java reflection.
2) No direct methods to algebraic operation functions are performed, `unaryOperation` and `binaryOperation` routines are called each time.
3) Direct calls are inserted but only in the case if the user provides a dictionary, which maps `Algebra` operation identifiers to method signatures.
4) `unaryOperationFunction` and `binaryOperationFunction` are used, and the functions they return are stored within the expression object.

There is a table of the advantages and disadvantages for each of the four options—I.

There were two attempts to implement Java bytecode generation.

The first one used the reflection lookup; however, it was considered non-universal because it is broken for cases when the operation name doesn't match the name of the function, or the function with that name doesn't perform the same thing as the operation. Implementing this algorithm was also

TABLE I
OPERATIONS CALLING APPROACHES ON JVM

	Reflection lookup	Direct dynamic calls	Method calls by the table	Indirect dynamic calls
Boxing problem	Only return value is boxed	Both arguments and return value are boxed	Only return value is boxed, or everything is boxed	Both arguments and return value are boxed, but some optimizations will be possible
Fails if operation name doesn't match the method name	Yes	No	No	No
An extra parameter should be passed to the compiler	No	No	Yes	No
Performs tableswitch operations lookup	Only if method can't be found	At each call	No	Only at compilation

cumbersome because many stack values had to be coerced, reflection had been used actively.

In the second attempt, the third option was accomplished—function type objects were collected from `unaryOperationFunction` like methods. This algorithm causes a larger boxing allocation overhead; however, it is much more stable and universal.

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

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

Fig. 1. Legacy bytecode generation result (decompiled)

Let us compare the bytecode generated by both generation algorithms in decompiled forms: from the legacy one—fig. 1, and from modern one—fig. 2.

Both classes are generated for expression $x + 2$ in the context of `RealField` that implements `ExtendedField<Double>`, so both classes implement `Expression<Double>`. The access flags, class name pattern, declared methods, and type signatures (except the key of arguments map is changed to `Symbol` from `String`, but it does not matter) are the same. The first difference is stored fields. Old bytecode generator emitted either one or two fields: the first one always stores a reference to `Algebra` object, the second one stores constants (as `Object[]`) required by the expression if they can't be placed to the class file's constant pool. New generator emits only one field—`constants`, which stores dynamic constants as well as Kotlin function objects produced by `unaryOperationFunction` and `binaryOperationFunction` methods. The second

```
import java.util.*;

import kotlin.jvm.functions.*;
import kscience.kmath.asm.internal.*;
import kscience.kmath.expressions.*;

public final class AsmCompiledExpression_45045_0
        implements Expression<Double> {
    private final Object[] constants;

    public AsmCompiledExpression_45045_0(
            Object[] constants) {
        this.constants = constants;
    }

    public final Double invoke(
            Map<Symbol, ? extends Double> arguments) {
        return (Double) ((Function2) constants[0])
                .invoke((Double) MapIntrinsics
                        .getOrFail(arguments, "x"), 2);
    }
}
```

Fig. 2. Current bytecode generation result (decompiled)

difference is the way of constructing the expression sequence: the old code generation has `Algebra` reference as receiver of all the operations, the new one has elements of the constants array only. Both generated classes are constructed with reflection.

VI. JAVASCRIPT SOURCE CODE GENERATION

Applying Kotlin Multiplatform to the created library it was easy to port MST features to Kotlin/JS (Kotlin for JavaScript) and Kotlin/Native (Kotlin for Native) as well as a feature similar to JVM dynamic compilation.

The development of JavaScript was straightforward. The idea of storing functions instead of `Algebra` references was derived from the Java bytecode backend—the generated function gets the similar constants array which stores both constant values of expression and function references of operations used in the expression.

As for tooling, estree [9] as JavaScript AST classes package and astring [10] as code generation framework were selected, and the only implementation choice was between creating sources by appending fragments to a string or building AST then rendering it.

The MST to JS compiler generates a function then wraps it as a KMath `Expression`. There is an example of such a function in fig. 3.

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

Fig. 3. Example of generated JavaScript function

VII. WebAssembly IR Generation

WebAssembly (or WASM) [11] is an open standard defining a portable IR for executable programs, and in the context of this study, WebAssembly code generation was considered. However, this way of compiling was much more limited than the generation of JVM bytecode and JS source code.

Since the compilation has to be dynamic, the Kotlin/JS was used for the prototype of this backend, and here are the concrete trade-offs and problems with it:

1) This compilation couldn't be universal because calling JS from WASM interoperability is confirmed to be slow hence Kotlin/JS builtin mathematical functions (which are simply delegated to JavaScript `Math` object, of course) were not available as well as any opportunities to invoke KMath contexts functions.
2) Only `f64` and `i32` WebAssembly types were supported because i64 isn't available without experimental V8 feature that maps `i64` to JavaScript `bigint` type. `f32` was not available for a similar reason—JavaScript doesn't have a type for single-precision floating-point format.
3) Basic mathematical functions for `f64` (like `sin` and `cos`) required to support operations that `RealField` does were taken from `libm` (also known as `math.h` [12]) which was compiled to WebAssembly and appended partially to the initial state of WebAssembly module. All the other `f64` arithmetic is available with WebAssembly opcodes.

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

Fig. 4. Example of emitted WASM IR in the WAT form

This backend uses `binaryen` [13] library to simplify IR generation and perform some optimizations.

Actually, the upcoming Kotlin/WASM (Kotlin for WebAssembly) target would be much more suitable because of the lack of interoperability overhead, but it is in early development phase now.

VIII. LLVM IR Generation

LLVM (Low-level Virtual Machine) [14] compiler infrastructure project is a set of compiler and toolchain technologies designed around an IR that serves as a portable, high-level assembly language.

LLVM was used as the backend of Kotlin/Native, so it has been investigated as a possible expression compilation target, too.

However, the decision has been made to forgo this feature for two reasons:

1) This generation target couldn't be universal because Kotlin/Native as host platform is hard to interoperate.
2) More importantly LLVM is huge and has pitiful compilation performance especially with higher optimization levels, so it is unsuited for dynamic compilation, and it made no point in carrying the LLVM at least for primitive typed computations as for WebAssembly. All the previously mentioned IRs have a lightweight or built-in language runtime generation infrastructure.

IX. Benchmark Results

The new expression APIs were microbenchmarked. Two measurements are included in this paper.

Environment data:

- CPU: Intel Core i5 6400, 3.196 GHz, Skylake
- RAM: 15.977 GiB
- OS: Ubuntu 20.10 Groovy

All the tested expressions API implementations calculate the formula below one million times by using double-precision floating-point arithmetic:

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

Java runtime:

1) OpenJDK Hotspot (build 11.0.10+9-LTS)
2) OpenJDK GraalVM CE 21.0.0 (build 11.0.10+8-jvmci-21.0-b06)

JMH [15] (Java Microbenchmark Harness) shipped within the kotlinx-benchmark [16] tool was used in throughput mode with 5 warm-ups and 5 plain iterations.

JVM hosted measurements are presented in table II.

TABLE II
JVM Hosted Measurements

	Description	Average frequency, Hz	
		Hotspot	**GraalVM**
functional	Functional expression	2.600	**4.003**
mst	Interpreted MST expression	0.118	**0.177**
asm	ASM compiled expression	2.994	**4.196**
raw	Raw implementation compiled statically	4.102	**7.864**

JS runtime:

1) NodeJS 12.16.1 (V8 7.8.279.23-node.31)

JS hosted measurements are presented in table III.

TABLE III
JS HOSTED MEASUREMENTS

	Description	Single shot time, s
		NodeJS
functional	Functional expression	3.61
mst	MST expression	254
wasm	WASM compiled expression	4.22
estree	ESTree compiled expression	**3.55**
raw	Statically written expression in Kotlin	5.47

X. CONCLUSION

The research on the dynamical interpretation and code generation for generic algebras in KMath is a work in progress. There are a lot of things to be done about performance optimization and API clean-up. Still, even current results show the principal possibility of using dynamic expression building even for performance-critical parts. The ASM code generation did not provide a significant performance boost but still is useful for research.

MST representation was a side-product of this research, but proved to be a valuable tool of its own. Since it is a syntactic tree with the possibility to support symbols, it is possible to use it for simple symbolic computations. For example, there is experimental support for automatic differentiation based on Kotlin∇ [5].

XI. ACKNOWLEDGMENT

The author would like to thank the members of the KMath development team (Alexander Nozik, Peter Klimai, and Roland Grinis) and Breandan Considine for discussion and revision of the work.

The KMath project is developed in cooperation between MIPT and JetBrains Research.

REFERENCES

[1] "Languages, Compilers, and Runtime Systems," [Online]. Available: https://www.eecs.umich.edu/eecs/research/area.html?areaname=languages-compilers (visited on 02/15/2021).

[2] "Kotlin Programming Language," [Online]. Available: https://kotlinlang.org (visited on 02/11/2021).

[3] A. Nozik, "Kotlin language for science and kmath library," *AIP Conference Proceedings*, vol. 2163, no. 1, p. 040 004, Oct. 2019. DOI: 10.1063/1.5130103. eprint: https://aip.scitation.org/doi/pdf/10.1063/1.5130103. [Online]. Available: https://aip.scitation.org/doi/abs/10.1063/1.5130103 (visited on 02/15/2021).

[4] A. Nozik, I. Postovalov, P. Klimai, B. Considine, A. Mischenko, B. Mittelbach, A. Trifonov, A. Radke, A. Antipov, and N. Klimenko, *Mipt-npm/kmath: 0.2.0-dev-5*, version v0.2.0-dev-5, Jan. 2021. DOI: 10.5281/zenodo.4452686. [Online]. Available: https://doi.org/10.5281/zenodo.4452686.

[5] B. Considine, M. Famelis, and L. Paull, "Kotlin∇: A shape-safe dsl for differentiable programming," 2019.

[6] "ObjectWeb ASM," [Online]. Available: https://asm.ow2.io (visited on 02/11/2021).

[7] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java® Language Specification, Java SE 11 Edition*. Oracle Corporation, Sep. 2018. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se11/jls11.pdf (visited on 02/13/2021).

[8] "GraalVM," [Online]. Available: https://www.graalvm.org (visited on 02/13/2021).

[9] "ESTree," [Online]. Available: https://github.com/estree/estree (visited on 02/13/2021).

[10] "Astring," [Online]. Available: https://github.com/davidbonnet/astring (visited on 02/13/2021).

[11] A. Haas, A. Rossberg, D. Schuff, B. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," Jun. 2017, pp. 185–200. DOI: 10.1145/3062341.3062363.

[12] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, Dec. 2011, ch. 7.12, 683 (est.) [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.h

[13] "Binaryen," [Online]. Available: https://github.com/WebAssembly/binaryen (visited on 02/13/2021).

[14] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[15] "OpenJDK: jmh," [Online]. Available: https://openjdk.java.net/projects/code-tools/jmh/ (visited on 02/12/2021).

[16] "kotlinx-benchmark," [Online]. Available: https://github.com/Kotlin/kotlinx-benchmark (visited on 02/13/2021).