KOTLIN∇

A SHAPE-SAFE DSL FOR DIFFERENTIABLE FUNCTIONAL PROGRAMMING

Breandan Considine [1] **Liam Paull** [1] **Michalis Famelis** [2]

ABSTRACT

Kotlin is a statically-typed programming language with support for embedded domain specific languages, asynchronous programming, and multi-platform compilation. In this work, we present an algebraically-grounded implementation of forward- and reverse- mode automatic differentiation (AD) with shape-safe tensor operations, written in pure Kotlin. Our approach differs from existing AD frameworks in that Kotlin∇ is the first shape-safe AD library fully compatible with the Java type system, requiring no metaprogramming, reflection or compiler intervention to use. A working prototype is available at: https://github.com/breandan/kotlingrad.

1 INTRODUCTION

Prior work has shown it is possible to encode a deterministic context-free grammar as a *fluent interface* (Gil & Levy, 2016) in Java. This result was strengthened to prove Java's type system is Turing complete (Grigore, 2017). As a practical consequence, we can use the same technique to perform shape-safe automatic differentiation (AD) in Java, using type-level programming. A similar technique is feasible in any language with generic types. We use *Kotlin*, whose type system is less expressive, but fully compatible with Java.

Differentiable programming has a rich history among dynamic languages like Python, Lua and JavaScript, with early implementations including projects like Theano, Torch, and TensorFlow. Similar ideas have been implemented in statically typed, functional languages, such as Haskell's Stalin∇ (Pearlmutter & Siskind, 2008b), DiffSharp in F# (Baydin et al., 2015) and recently Swift (Wei, 2018). However, the majority of existing automatic differentiation (AD) frameworks use a loosely-typed DSL, and few offer shape-safe tensor operations in a widely-used programming language.

Existing AD implementations for the JVM include Lantern (Wang et al., 2018), Nexus (Chen, 2017) and DeepLearning.scala (Bo, 2018), however these are Scala-based and do not interoperate with other JVM languages. Kotlin∇ is fully interoperable with vanilla Java, enabling broader adoption in neighboring languages. To our knowledge, Kotlin has no prior AD implementation. However, the language contains a number of desirable features for implementing a native AD framework. In addition to type-safety and interoperability, Kotlin∇ primarily relies on the following language features:

1. **Operator overloading and infix functions** allow a concise notation for defining arithmetic operations on tensor-algebraic structures, i.e. groups, rings and fields.
2. **λ-functions and coroutines** support backpropogation with lambdas and shift-reset continuations, following Pearlmutter & Siskind 2008a and Wang et al. 2018.
3. **Extension functions** support extending classes with new fields and methods and can be exposed to external callers without requiring sub-classing or inheritance.

2 USAGE

Kotlin∇ allows users to implement differentiable programs by composing operations on fields to form algebraic expressions. Expressions are lazily evaluated inside a numerical context, which may imported on a per-file basis or lexically scoped for finer-grained control over the runtime behavior.

```
import edu.umontreal.kotlingrad.numerics.DoublePrecision

with(DoublePrecision) { // Use double-precision protocol
  val x = variable("x") // Declare immutable vars (these
  val y = variable("y") // are just symbolic constructs)
  val z = sin(10 * (x * x + pow(y, 2))) / 10 // Lazy exp
  val dz_dx = d(z) / d(x) // Leibniz derivative notation
  val d2z_dxdy = d(dz_dx) / d(y) // Mixed higher partial
  val d3z_d2xdy = grad(d2z_dxdy)[x] // Indexing gradient
  plot3D(d3z_d2xdy, -1.0, 1.0) // Plot in -1 < x,y,z < 1
}
```

Figure 1. A basic Kotlin∇ program with two inputs and one output.

Above, we define a function with two variables and take a series of partial derivatives with respect to each variable. The function is numerically evaluated on the interval $(-1, 1)$ in each dimension and rendered in 3-space. We can also plot higher dimensional manifolds (e.g. the loss surface of a neural network), projected into four dimensions, and

[1]Mila, Université de Montréal [2]Université de Montréal. Correspondence to: Breandan Considine <bre@ndan.co>.

Proceedings of the 2^{nd} SysML Conference, Palo Alto, CA, USA, 2019.

rendered in three, where one axis is represented by time. To demonstrate, a display is required for animation purposes.

$$z = \sin\left(10(x * x + y^2)\right)/10, \texttt{plot}\left(\frac{\partial^3 z}{\partial x^2 \partial y}\right)$$

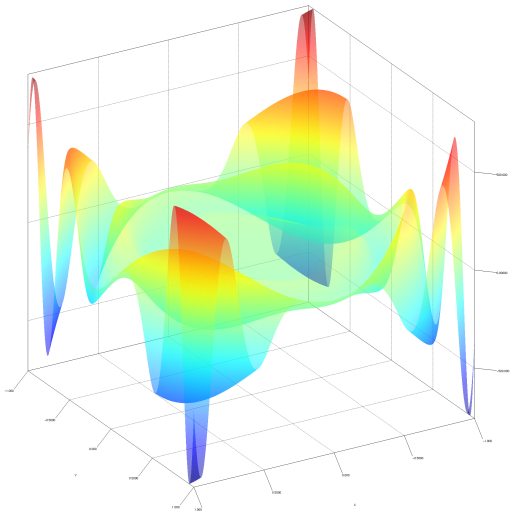

Figure 2. Output generated by the program shown in Figure 1.

Kotlin∇ treats mathematical functions and programming functions with the same underlying abstraction. Expressions are composed recursively to form a data-flow graph (DFG). An expression is simply a `Function`, which is only evaluated once invoked with numerical values, e.g. `z(0, 0)`.

```
val z = sin(10 * (x * x + pow(y, 2))) / 10
```

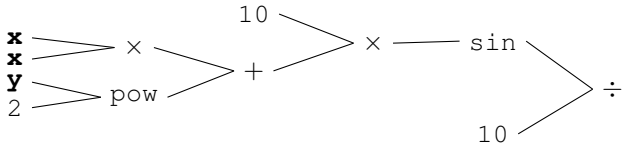

Figure 3. Implicit DFG constructed by the original expression, `z`.

Kotlin∇ supports shape-shafe tensor operations by encoding tensor rank as a parameter of the operand's type signature. By enumerating type-level integer literals, we can define tensor operations just once using the highest literal, and rely on Liskov substitution to preserve shape safety for subtypes.

```
// Literals have reified values for runtime comparison
open class `0`(override val value: Int = 0): `1`(0)
open class `1`(override val value: Int = 1): `2`(1)
class `2`(open val value: Int = 2) // Greatest literal
// <L: `2`> will accept L <= 2 via Liskov substitution
class Vec<E, L: `2`>(len: L, cts: List<E> = listOf())
// Define addition for two vectors of type Vec<Int, L>
operator fun <L: `2`, V: Vec<Int, L>> V.plus(v: V) =
  Vec<Int, L>(len, cts.zip(v.cts).map { it.l + it.r })
// Type-checked vector addition with shape inference
val Y = Vec(`2`, listOf(1, 2)) + Vec(`2`, listOf(3, 4))
val X = Vec(`1`, listOf(1, 2)) + Vec(`3`) // Undefined!
```

Figure 4. Shape safe tensor addition for rank-1 tensors, $\forall L \leq 2$.

It is possible to enforce shape-safe vector construction as well as checked vector arithmetic up to a fixed `L`, but the full implementation is omitted for brevity. A similar pattern can be applied to matrices and higher rank tensors, where the type signature encodes the shape of the operand at runtime.

With these basic ingredients, we have almost all the features necessary to build an expressive shape-safe AD, but unlike prior implementations using Scala or Haskell, in a language that is fully interoperable with Java, while also capable of compiling to JVM bytecode, JavaScript, and native code.

In future work, we intend to implement a full grammar of differentiable primitives including matrix convolution, control flow and recursion. While Kotlin∇ currently implements arithmetic manually, we plan to wrap a BLAS such as cuBLAS or native linear algebra library for performance.

References

Baydin, A. G., Pearlmutter, B. A., and Siskind, J. M. Diffsharp: Automatic differentiation library. *CoRR*, abs/1511.07727, 2015. URL http://arxiv.org/abs/1511.07727.

Bo, Y. DeepLearning.scala: A simple library for creating complex neural networks. 2018. URL https://github.com/ThoughtWorksInc/DeepLearning.scala.

Chen, T. Typesafe abstractions for tensor operations (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, pp. 45–50, 2017. doi: 10.1145/3136000.3136001. URL http://doi.acm.org/10.1145/3136000.3136001.

Gil, Y. and Levy, T. Formal language recognition with the java type checker. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

Grigore, R. Java generics are Turing Complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pp. 73–85, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009871. URL http://doi.acm.org/10.1145/3009837.3009871.

Pearlmutter, B. A. and Siskind, J. M. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):7, 2008a.

Pearlmutter, B. A. and Siskind, J. M. Using programming language theory to make automatic differentiation sound and efficient. pp. 79–90, 2008b.

Wang, F., Wu, X., Essertel, G. M., Decker, J. M., and Rompf, T. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *CoRR*, abs/1803.10228, 2018. URL http://arxiv.org/abs/1803.10228.

Wei, R. First-class automatic differentiation in Swift: A manifesto. 2018. URL https://gist.github.com/rxwei/30ba75ce092ab3b0dce4bde1fc2c9f1d.