International Journal of
Computer Science and
Programming Language

Volume 7, Issue 2, 2021
DOI (Journal): 10.37591/IJCSPL

http://computers.journalspub.info/index.php?journal=JCSPL&page=index

Review IJCSPL

Exploring the Potential of the Kotlin Programming Language for Scientific Computing with the KotlinLab Environment

Stergios Papadimitriou*

Abstract

KotlinLab is an easy to use MATLAB-like environment for the Java Virtual Machine (JVM). It implements scripting based on the Kotlin programming language. This paper shows that Kotlin Lab can significantly accelerate and simplify the development of complex scientific computer software in the JVM. The work compares Java based approaches with Kotlin ones, and illustrates the significant advantages of the enhanced flexibility of the Kotlin language. We illustrate that one of the more significant drawbacks of Java is its lack of operator overloading. Also, the nominal approach to functional programming of Java is not as flexible and convenient as the structural approach of Kotlin is. The paper presents the architecture of the KotlinLab environment, and the way that it exploits effectively the Kotlin JSR223 scripting implementation. The main concern of the KotlinLab enviroment is to exploit the potential of the Kotlin language in order to provide a rich and easy to use scientific programming environment. In this way the paper tries to highlight some important and advanced characteristics of the Kotlin language that provide important benefits to the programming experience. In particular, the Kotlin language is especially effective in building DSLs (Domain Specific Languages). At this domain, the paper illustrates the straightforward way to implement scientific operators.

Keywords: Domain-Specific Languages (DSLs), Java Virtual Machine (JVM), Scripting, MATLAB, Functional programming, Object-oriented programming.

INTRODUCTION

KotlinLab is a MATLAB like environment for scientific computing that exploits the potential of the Java Virtual Machine. It utilizes the Kotlin programming language [1–5].

KotlinLab is an open-source project and can be obtained from https://github.com/sterglee/KotlinLabSimple. The architecture of KotlinLab is shown in Figure 1. KotlinLab presents similarities with ScalaLab [6–9] both in architecture and functionality. Kotlin as Scala also, even are statically typed language, provide extensive support for building Domain Specific Languages (DSLs) [10–4]. Although these languages are statically typed, they are designed to provide syntax fluency. Also, the Kotlin language supports a powerful functional programming environment that competes well with Scala.

The paper proceeds as follows: Section 2 elucidates the implementation of scientific

***Author for Correspondence**
Stergios Papadimitriou
E-mail: sterg@cs.ihu.gr

Professor, Department of Computer Science, International Hellenic University, 65404 Kavala, Greece

Received Date: December 07, 2021
Accepted Date: December 13, 2021
Published Date: December 17, 2021

Citation: Stergios Papadimitriou. Exploring the Potential of the Kotlin Programming Language for Scientific Computing with the KotlinLab Environment. International Journal of Computer Science and Programming Language. 2021; 7(2): 11–17p.

scripting within KotlinLab. Specifically, it describes how KotlinLab exploits the Kotlin scripting programming interface to build upon it a higher-level MATLAB like scripting. Section 3 discusses many aspects concerning on the exploitation of advanced Kotlin language characteristics within KotlinLab. Finally, Section 4 presents the conclusions along with directions for future work.

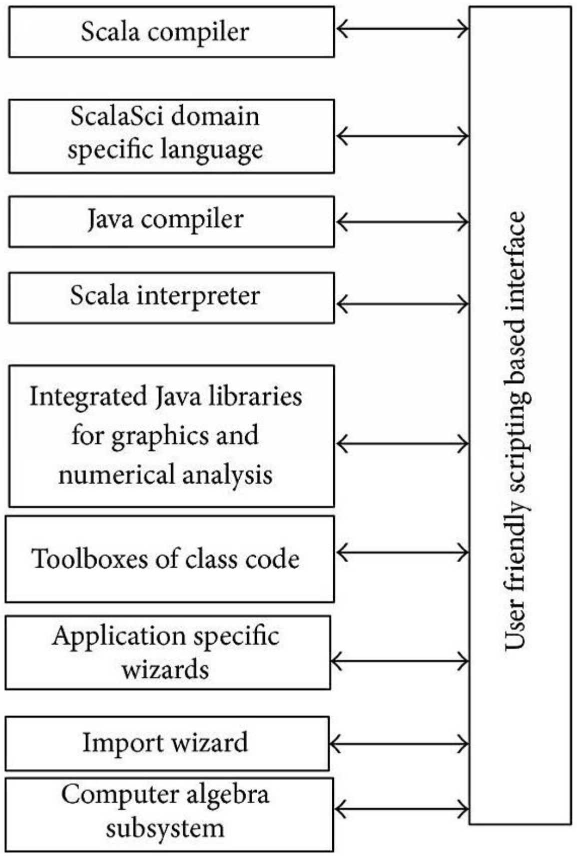
Scala compiler
ScalaSci domain specific language
Java compiler
Scala interpreter
Integrated Java libraries for graphics and numerical analysis
Toolboxes of class code
Application specific wizards
Import wizard
Computer algebra subsystem
User friendly scripting based interface

Figure 1. The architecture of KotlinLab.

SCIENTIFIC PROGRAMMING WITH KOTLIN BASED SCRIPTING IN KOTLINLAB

Responding to the demand of the users, modern languages implement powerful REPLs (Read-Evaluate-Print Loop) tools. Kotlin is no exception, and it has both a default embedded REPL and a more elaborate one, the KShell [https://github.com/Kotlin/kotlin-interactive-shell] that is developed as an open source project. Both REPLs are based on compiling the scripts (using the Kotlin compiler API) instead of interpreting them. Therefore, scripts can run at full speed, something of much importance for scientific programming. The default REPL provides a standard JSR 223 scripting interface, and therefore its API is standardized and easy to use. The KShell is more advanced and it provides strong facilities as code completion and syntax highlighting. It's adaptation within KotlinLab is in development.

The default Kotlin Shell ships together with the standard Kotlin installation, something that fits well for rapid prototyping of code. However, there is also full support for the JSR 223 standard scripting interface API [12] for utilizing the potential of scripting within the context of other applications. KotlinLab exploits this API to bring to the user the potential to experiment interactively with Java scientific libraries. There are a lot of libraries that KotlinLab exploits effectively some described in references [13–17].

The JSR 223 API provides programmatic access to the script evaluation engine of the Kotlin REPL that provides significant flexibility and power. For example, import statements are persistent. Also persistent are the class, functions and methods declarations. The persistence of definitions (e.g. imports, methods, class definitions) is very convenient. Error handling can provide specific and elaborate messages and variable and method declarations and redefinitions are flexible.

The JSR223 standard is based on a *ScriptEngineManager* object to obtain a *ScriptEngine* object. In KotlinLab the *ScriptEngine* object corresponds to the Kotlin's scripting engine. Each scripting engine has a set of *bindings* that implement the workspace of globally accessible variables, realizing the concept of the state of a scripting session. These state variables are modified by the results of the execution of scripts. Additionally, they can be set/get with simple API calls. Also, the evaluation of scripts is particularly easy, by means of a simple *eval()* call.

KotlinLab prepares the REPL instance by appending automatically the jar files of all of its important libraries as well as the additional user installed toolboxes. In order to facilitate the user further, KotlinLab provides a folder named *extra Jars for Kotlin Classpath.* Any jar file placed at this folder, is automatically appended at the Kotlin's REPL classpath. The user also can control the classpath with graphical user interface options in order to add/remove dynamically components.

Kotlin scripting implements several strong characteristics. The flexibility of the Kotlin language permits the implementation of higher level syntactic constructs. For example, in Kotlin it is easy to express multiplication of matrices *A* and *B* as *A*B*, while Java enforces to use a method call like e.g. *A.mult(B).* Kotlin, although is statically typed, provides extensive syntax fluency and competes well on DSL building potential with other syntactical fluent languages as Groovy and Scala.

CHARACTERISTICS OF KOTLIN FOR EFFICIENT SCIENTIFIC PROGRAMMING

Kotlin provides a lot of powerful new features while at the same time it maintains superb Java interoperability facilitating greatly the exploitation of the vast Java libraries of scientific code. This section aims to highlight some of these features.

The Kotlin compiler uses primitive types when non-nullable Java primitive types are used and therefore the scientific code that actually operates on primitive types is as efficient as the corresponding Java code is.

Many JVM languages (e.g. Scala) implement their own collection frameworks. This approach however imposes Java interoperability problems, since extra conversions are required, implying additional overhead and complexity of code. Since Java already has a powerful collection library, Kotlin designers choosen to exploit and improve it, rather than to replace. Kotlin's collections are compile-time views [18] and map directly to JDK collections. Therefore, there is no runtime overhead to use the Kotlin collections API to interact with Java.

Kotlin handles ranges efficiently by compiling them into index-based for loops, that are supported by the JVM, thereby avoiding the performance penalty from creating iterator objects. For-loops work in Kotlin seemingly for Java collections. Also, the compiler has special support for arrays, that compile a loop over an array to a normal index-based for loop, thus avoiding any performance penalty. Kotlin allows efficient lazy evaluation of collections. Any collection can be converted to a *sequence* for maximizing the performance. Kotlin Sequences resemble the operation of the Streams of modern Java.

Kotlin is far superior from Java at the potential for syntax fluency and for creating DSLs, and some clever design decisions can place the gained flexibility sometimes even farthest from Scala. For example, one such decision is that Kotlin avoids the new keyword for creating classes. To create an instance of a class, we call the constructor as if it were a regular function, therefore facilitating more succint syntax of Domain Specific Languages (DSLs).

Kotlin implements operator overloading by mapping operators to specially named methods. For example, the operator symbol * corresponds to a call to a *times()* method. Especially important, is the potential of Kotlin to overload operators on third party classes by using the extension functions [2–4].

A useful characteristic of Kotlin's overload system, is that when we define an operation for an operator such as plus, Kotlin supports not only the + operation but += as well. Operators such as +=, -= and so on, are called compound assignment operators.

Kotlin implements the convenient [] operator (i.e. indexing operator) for retrieving values from collections, nicely supporting the [] syntax for any object of a class that offers an one-parameter *get()* function. The compiler translates the operator [], with the corresponding *get()* method call. The language also provides flexible indexing operators that can access objects with the operator []. Clearly, for one dimensional arrays, the indexing is performed with ints. The retrieval of an element corresponds to a call to the *get(int index)* function. Furthermore, the concept is generalized to multi-dimensional arrays and to index using arbitrary indexes (i.e. not necessarily ints).

A significant aspect of Kotlin concerning also Java interoperability is that operator conventions work as well for code implemented in Java. For example, the *Matrix* class that implements functionality with a Java 2-D array is a Java class and implements a *times()* method for matrix multiplication. This method is exposed to Kotlin via the operator overloading framework and very nicely the Kotlin code can perform matrix multiplication with the * operator (in addition to the *times()* method call).

Kotlin permits to inject methods and properties into any JVM class. Also, unlike dynamically typed languages such as Groovy [19] that patch the runtime, Kotlin performs these metaprogramming operations at compile time, thereby performance is not compromised. *Extension functions* and *extension properties* are the techniques used to add methods and properties, without altering the bytecode of the targeted classes. Existing instance methods of a class take precedence over extension methods in the case of conflict. We can inject methods and properties into existing classes, including final classes.

The example code below demonstrates how extension functions are implemented for the *Mat1D* class, in order for example to be able to write expressions as 5.5*A, where A is a matrix object of the Mat1D class.

```
package kotlinLabSci.math.array
// this object defines the extension functions
object ExtensionsMat1D {
 operator fun Double.plus(x: Mat1D):Mat1D = x.plus(this)
 operator fun Int.plus(x: Mat1D):Mat1D = x.plus(this.toDouble())
 operator fun Double.minus(x: Mat1D):Mat1D = x.minus(this)
 operator fun Int.minus(x: Mat1D): Mat1D = x.minus(this.toDouble())
 operator fun Double.times(x: Mat1D): Mat1D = x.times(this)
 operator fun Int.times(x: Mat1D): Mat1D = x.times(this.toDouble())
}
```

Kotlin in addition to methods permits to inject also properties. These properties can't use backing fields since they do not have access to the internals of the class.

Kotlin doesn't require classes, and we can work with top-level functions when they suffice. In Kotlin, we can place functions directly at the top level of a source file, outside of any class. This facilitates the handling of functions, which are the basic building block of scientific code. Scientists usually experiment with the execution of small scripts, and therefore this feature is valuable.

In terms of generic programming, Kotlin provides also more tools and additional flexibility compared to Java. Specifically, it supports also *declaration site variance* in addition to *use-site variance* (Java supports only use-site variance). With declaration site variance we specify the variance for a specific occurrence of a type parameter once at the declaration and thus the code becomes much more concise and elegant. However, use site variance is generally more flexible. Kotlin permits the specification of both use-site variance and of declaration site variance, thereby facilitating the implementation of generics libraries. For example, with interfaces that both consume and produce values of type T, T should be generally invariant, i.e. we can't apply declaration site variance. However, we can apply use-site variance for specific cases, for example, parameters that are not modified by functions can be set as covariant.

Inline classes permit many times to avoid the overhead of object creation and handling. They offer the benefit of classes with the enforced type checking by the compiler at compile time and the runtime efficiency of primitive types. Calls to inline functions are replaced by the Kotlin compiler with their body, avoiding the overhead of a function call.

An important advanced characteristic of the Kotlin compiler is that it can overcome the limitations of JVM's type erasure for *inline* functions by using *type reification* [2, 4, 5, 6, 11]. Every time that we call a function with a reified type parameter, the compiler knows the exact type used as the type argument in that particular call. This permits the compiler to generate efficient bytecode that uses the specific types. We should note that marking functions as inline has significant performance benefits. When these functions have parameters of the function type and the corresponding arguments (i.e. lambdas) are inlined together with the functions.

One significant and rather peculiar and distinct characteristic of Kotlin is its way to confront the null pointer dereferencing error and to prevent it by detecting that at compile time. Kotlin distinguishes *nullable* types for which null is a valid value from *non-nullable* types that cannot. For the later, the compiler rises a compile time error whenever it detects a possible assignment to null value. Besides the integration of nullability within type system of the language a lot of nice conveniency operators are provided as then safe call operator, the Elvis operator, the smart casts etc., and the overall setup forwards the confrontation of nullability issues to a level beyond most other competetive languages.

Kotlin implements a full conformed JSR223 scripting engine. In contrast, the JShell of modern Java [20–22], although it has a rich programming interface, it is significantly different from the JSR223 standard [23]. Clearly this is inconvenient since additional time to become familiar with the interface from the programmers is required from the Java programmer.

Concerning Kotlin scripting, a very useful property is the *Kotlin.script.classpath* that can be used to set the classpath of the Kotlin scripting engine. KotlinLab exploits the API interface support for JAR file manipulation that Kotlin provides and reads the *ClassPath* from the *MANIFEST* file of the KotlinLab jar file. The basic classpath constitutes the compile time classpath, i.e. it is consisted from all the libraries that are necessary to compile the source code of KotlinLab. This classpath is extended with the additional toolboxes that are installed in KotlinLab. Thereafter, all the KotlinLab's scientific libraries are accessible from the classpath of the KotlinLab's engine.

Functional programming can elegantly pack a lot of actions into a few lines of code and is highly expressive for many scientific computation tasks. Kotlin has function literals and functional types that facilitate the implementation of functional code. Functional types smoothly integrate functions into the language type system with better expressiveness than Java. These functional types facilitate greatly the applicability of functional programming. In contrast the Java's nominal approach requires to formulate the application in terms of SAM (Single Abstract Method) functional interfaces that can be implemented conveniently with lambda expressions.

Also, Kotlin lambdas are more flexible since they can modify captured variables from their context. In addition, Kotlin callable references are first-class expressions. In contrast, Java's method references only make sense in the context of some functional interface [24].

Recently, Java introduced immutable versions for its collections in order to better support functional programming [25]. However, both mutable and immutable versions of Java's collections implement the same interfaces. As a result the call to a method that mutates an immutable object is acceptable by the compiler but it raises an *UnsupportedOperationException* at runtime. Kotlin improves on that with the elegant views feature that permits the compile time detection of such errors. Kotlin provides two different views for lists, sets and maps: the immutable view and the mutable one. Both views do not impose any overhead since they map to the underlying Java collections with compile time manipulation [26].

All collection manipulating operations are inline functions. Therefore, the ease of their use doesn't involve any performance penalties related to function calls and lambdas.

CONCLUSIONS AND FUTURE WORK

The paper has explored the potential of the new Kotlin programming language with the KotlinLab MATLAB-like environment framework. The design of KotlinLab centered around the powerful features and the flexibility of the Kotlin language was elucidated.

The paper has demonstrated that KotlinLab builds an effective and easy to use scientific programming environment by exploring the syntactic fluency and the strong object-functional features of Kotlin. JVM is a superb development framework that is based on advanced technology and algorithms. The advent of improved JVMs supporting more efficiently scientific computation tasks will enrich furtrher the potential of KotlinLab for high-performance computing.

REFERENCES

1. Stephen Samuel and Stefan Bocutiu, Learn Kotlin Programming, Packt 2019. https://www.packtpub.com/product/learn-kotlin-programming-second-edition/9781789802351
2. Dmitry Jemerov, Svetlana Isakova, Kotlin in Action, Manning 2017. https://www.manning.com/books/kotlin-in-action
3. Ken Kouse, Kotlin Cookbook, O'Reily, 2018. https://www.oreilly.com/library/view/kotlin-cookbook/9781492046660/
4. Pierre-Yves Saumont, The Joy of Kotlin, Manning, 2019. https://www.manning.com/books/the-joy-of-kotlin
5. Marcin Moskala, Effective Kotlin, Leanpub, 2020. https://leanpub.com/effectivekotlin
6. Stergios Papadimitriou, Konstantinos Terzidis, Seferina Mavroudi, Spiridon Likothanasis, "ScalaLab: an effective scientific programming environment for the Java Platform based on the Scala object-functional language", IEEE Computing in Science and Engineering (CISE), Vol. 13, No 5, 2011, p. 43–55
7. Papadimitriou S., Mavroudi S., Theofilatos K., Likothanasis S., The software architecture for performing scientific computation with the JLAPACK libraries in ScalaLab, Scientific Programming, 2012; 20 (4), pp. 379–391
8. Stergios Papadimitriou, Lefteris Moussiades. Combining Scala with C++ for efficient scientific computation in the context of ScalaLab, Lecture Notes in Engineering and Computer Science. 2016; 1: 409–412

9. Papadimitriou S, Moussiades L, The design of JVM and native libraries in ScalaLab for efficient scientific computation, International Journal of Modeling, Simulation and Scientific Computing 9 (5), 1850037, 2018
10. Gilles Dubochet, "On Embedding domain-specific languages with user-friendly syntax", In Proceedings of the 1st Workshop on Domain Specific Program Development, pages 19–22, 2006
11. Gilles Dubochet, "Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming", PhD Thesis, EPFL, Suise, 2011
12. Cay Horstmann, Gary Cornell, Core Java 2, Vol I Fundamentals, Vol II-Advanced Techniques. Sun Microsystems Press, 11th edition, 2019
13. Hang T. Lau, A Numerical Library in Java for Scientists and Engineers, Chapman & Hall/CRC, 2003
14. E. Anderson, Z. Bai, C. Birschof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. Mckenney, D. Sorensen, LAPACK Users' Guide, SIAM, Third Edition, 1999
15. Timothy A. Davis, "Direct Methods for Sparse Linear Systems", SIAM publishing, 2006
16. Kazushige Goto, Robert A. Van De Geijn, Anatomy of High-Performance Matrix Multiplication, ACM Transactions on Mathematical Software, Vol[. 34, No 3, Article 12, May 2008
17. Field G.Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. Van de Geijn, Francisco D. Igual, Michail Smelyankiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, Lee Killough, The BLIS framework: Experiments in portability, ACM Transactions on Mathematical Software, 42 (2), 12, 2016
18. Venkat Subramaniam, Programming Kotlin: Create Elegant, Expressive, and Performant Jvm and Android Applications. O′Reilly (20 September 2019).
19. Dierk Konig, Andrew Glover, Paul King, Guillaume Laforge, Jon Skeet, Groovy In Action, Manning Publications, 2007
20. Mala Gupta, Java 11 and 12–New Features: Learn about Project Amber and the latest developments in the Java language and platform. Packt Publishing Limited (26 March 2019).
21. Edward Laviei, Mastering Java 11-Second Edition. 2nd Ed. 2018. https://www.packtpub.com/product/mastering-java-11-second-edition/9781789137613
22. Nick Samoylov, Learn Java 12 Programming, Packt 2019. https://www.packtpub.com/product /learn-java-12-programming/9781789957051
23. Kishori Sharan, Java 9 Revealed, Apress 2017. https://link.springer.com/book/10.1007/978-1-4842-2592-9
24. Richard Warburton, Java 8 Lambdas: Functional Programming for The Masses. Shroff/O'Reilly; First edition (1 January 2014).
25. Xiao-Feng Li, Advanced Design and Implementation of Virtual Machines, CRC Press, 201
26. Aleksei Sedunov, Kotlin In-depth [Vol-II]: A comprehensive guide to modern multi-paradigm language (English Edition). BPB Publications; 1st edition (7 March 2020).