Multilanguage Frameworks

Towards the Ideal Multilanguage Environment

Julius Hřivnáč[1,*]

[1]Université Paris-Saclay, CNRS/IN2P3, IJCLab, 91405 Orsay, France

Abstract. High Energy Physics software has been a victim of the necessity to choose one implementation language as no really usable multi-language environment existed. Even a co-existence of two languages in the same framework (typically C++ and Python) imposes a heavy burden on the system. The role of different languages was generally limited to well encapsulated domains (like Web applications, databases, graphics), with very limited connection to the central framework.
The new development in the domain of the compilers and run-time environments has enabled ways for creating really multilanguage frameworks, with seamless, user-friendly and high-performance inter-operation of many languages, which traditionally live in disconnected domains (like C-based languages vs JVM languages or Web languages).
Various possibilities and strategies for creation of the true multi-language frameworks are discussed, emphasizing their advantages and possible road blocks.

1 An Ideal Multilanguage Application

In the pursuit of creating an ideal multilanguage application, the primary goal is to leverage the strengths of various programming languages and tools seamlessly, promoting efficient and effective development. Such an application would exhibit the following characteristics:

- **Utilizing the Best Tools and Languages:** The application should have the flexibility to choose the best programming languages and tools for each specific task. This approach allows developers to capitalize on the strengths and capabilities of each language, optimizing performance and productivity.
- **Transparent Interfaces, No Stubs:** The interfaces between different components or modules of the application should be transparent, without the need for stubs or intermediary layers. This transparency ensures that data and functionality flow seamlessly between different parts of the application.
- **Data Sharing, No Proxies:** Data sharing should occur directly, without the need for proxies or complex data transformation layers. This approach simplifies data exchange and minimizes the risk of data loss or errors during translation.
- **Seamless Functionality:** The application should works seamlessly. All components, regardless of the languages they are written in, should collaborate harmoniously to deliver the intended functionality.

*e-mail: Julius.Hrivnac@cern.ch

The vision of an ideal multilanguage application is no longer a distant dream. With advancements in software development, interoperability, and the availability of powerful language-agnostic tools and platforms, this ideal is becoming a reality.

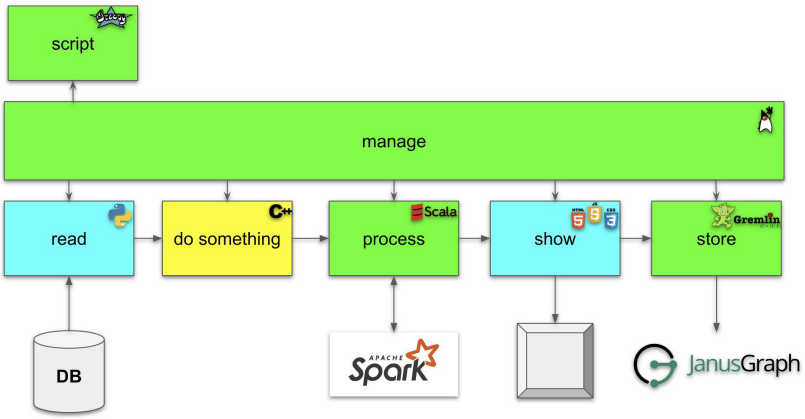

Figure 1. An example of an ideal multi-language architecture.

2 The general multilanguage technology status

2.1 JVM Languages

2.1.1 Java

Java, a high-level programming language, has established itself as a dynamic and flexible environment for software development. This robust ecosystem encompasses the Java Language, the Java Virtual Machine (JVM) for runtime execution, and a comprehensive set of standard libraries.

Java's dynamic nature allows for introspection, memory management, and boasts a wide array of monitoring and profiling tools, making it a versatile choice for various software development needs.

2.1.2 Java performance

When assessing Java's performance, it's important to consider its characteristics in relation to other programming languages and the types of tasks at hand. Java, like other languages, exhibits varying levels of performance depending on the nature of the operations.

Java excels in areas like object-oriented (OO) features, memory management, parallelism, and dynamic optimization. These strengths make it a preferred choice for a wide range of applications, including large-scale, enterprise-level systems.

However, there are scenarios where Java may lag behind other languages. For example, tasks involving matrix manipulations can be slower in Java, as it lacks native matrix support. Additionally, some numerical operations may incur a performance cost, which is often a trade-off for ensuring exact reproducibility of results. Java applications also typically have a slower startup time compared to natively compiled languages, as the Java Virtual Machine (JVM) needs to load and perform initial optimization.

Java applications tend to require more memory due to features like reflection, memory management, and dynamic capabilities, which enable runtime optimizations. Comparing Java's performance to other languages is a nuanced task. When benchmarking Java, it's essential to evaluate its performance in real-world applications, considering not only the language itself but also auxiliary functionality such as memory management, reflection, and parallelism.

2.1.3 Other Languages in the Java Ecosystem

In the expansive Java ecosystem, there exists a realm of languages that seamlessly interoperate with Java. These languages can coexist within the same runtime environment or compile into standard Java class files, enabling developers to use their unique features alongside Java.

The interoperability of these languages is not merely a superficial connection; it is a profound integration that allows for the free mixing of code, even via inheritance. This means that developers can combine elements from different languages within their projects, optimizing each language's strengths for specific tasks.

Furthermore, these languages offer developers the flexibility to use them in both scripting interpreted modes and compiled modes, adapting to the needs of the project at hand.

Java's commitment to evolution is reflected in its incorporation of successful features from these languages into its own ecosystem. For instance, Java has integrated functional syntax from languages like Scala, enriching its own capabilities.

- **Groovy (Apache):** Groovy is a highly expressive scripting language that operates at a very high level. It is particularly well-suited for scripting tasks and is employed in domains such as graph databases. [1]
- **Scala (Apache):** Scala is a functional programming language that brings powerful functional features to the Java ecosystem. Widely used in the context of Apache Spark, Scala offers expressive capabilities for large-scale data processing. [2]
- **Kotlin (Google):** Kotlin is an officially supported language for Android development, offering concise and expressive syntax. It has gained significant popularity in the Android development community. [3]
- **Clojure:** Clojure is a Lisp-like language that is known for its simplicity and elegance. It is designed for concurrent programming and is often used in scenarios where high concurrency is essential. [4]
- **BeanShell:** BeanShell provides interpreted and scripted Java capabilities. It allows developers to execute Java code interactively and is a valuable tool for various scripting tasks. [5]

Following is an example of the Groovy script converting an SQL table into an XML file. It can be either run as a shell script or compiled.

```
#!/usr/bin/env groovy
sql = Sql.newInstance("jdbc:mysql://localhost/Tuples",
                      "org.gjt.mm.mysql.Driver")
xml = new MarkupBuilder(new File("Tuples.xml"))
xml.tagSet() {
  sql.eachRow("select * from tuple where run > 2") {
    row -> xml.tag(Run:row.run, Event:row.event)
    }
  }
```

2.2 Managed languages

Managed languages within the Java ecosystem encompass a wide array of languages originating from diverse backgrounds. These languages achieve interoperability with Java through various means, such as re-implementation or the use of specific bridges and interfaces.

Among the multitude of managed languages that can coexist and interoperate with Java, some notable examples include Go, Haskell, JavaScript, Lisp, OCaml, Pascal, PHP, Python, R, Rexx, Ruby, Scheme, Smalltalk, Tcl, and many more. In fact, there are more than 100 languages available within the Java ecosystem, each offering unique features and capabilities.

2.3 C-world: Native Compilation

The C-world represents a domain of native code compilation, where languages are compiled directly to machine code. This process is sometimes facilitated by pre-compiling to C, which serves as an intermediary step in generating native code.

In contrast to managed languages like Java, the C-world often lacks high-level management features such as reflection and introspection. Developers typically implement these functionalities using in-house solutions, which can introduce compatibility issues and complexities.

While C-world languages are often perceived as faster and smaller in terms of executable size, these advantages come at a cost. One of the key drawbacks is the potential lack of functionality, which may limit the capabilities of programs. Additionally, achieving reproducibility and portability can be challenging in the C-world due to its low-level nature.

Higher-level concepts, which are readily available in managed languages, require complex implementations in the C-world. Connecting managed JVM languages with low-level C languages can be challenging and typically relies on direct interfaces like JNI (Java Native Interface) or JNA (Java Native Access) due to the unmanaged environment of C-world languages.

3 GraalVM: A New Managed Environment

GraalVM[6] represents a cutting-edge managed environment that breaks the barriers between different programming languages, offering support for both JVM-based and C-based languages. This innovative platform can execute code within a virtual machine (VM) or natively on the host system.

One of the defining features of GraalVM is its status as a universal VM, where non-JVM languages are placed on an equal footing with JVM languages. This level of parity enables full interoperability between languages, fostering seamless integration and interaction. Unlike traditional multi-language environments, where languages run side-by-side with frequent data conversions, GraalVM facilitates a harmonious coexistence of languages within the same VM.

GraalVM stands out for its exceptional speed and compact size, outperforming OpenJVM[7]. This advantage is partially attributed to GraalVM's implementation in Java, whereas OpenJVM is primarily written in C++. GraalVM achieves full interoperability with OpenJVM, allowing programs compiled for one environment to run in the other without hassle.

GraalVM is designed to be embedded in external applications, offering compatibility with popular software projects such as Oracle, Apache, MySQL, and more. It can even generate native executables and libraries using an Ahead Of Time (AOT) compiler, as opposed to the Just-In-Time (JIT) compilation commonly used in the JVM world. This approach results in

a smaller footprint, faster startup times, and, in some cases, accelerated execution, although it may entail a loss of certain dynamic features.

3.1 JIT vs AOT Compilation

When considering the generation of GraalVM native images, it becomes evident that this approach often outperforms rewriting Java code in languages like C, C++, or Go. This distinction lies in the compilation strategy employed.

JIT, or Just-In-Time Compiler, is the typical approach used by Java and other languages. It involves compiling code into bytecode (e.g., JAR files) and dynamically recompiling it at runtime by the Java Virtual Machine (JVM). JIT compilation is complex due to the highly dynamic nature of languages like Java, which attempt to understand runtime behavior. During compilation, it runs initialization code and creates the initial heap. AOT, or Ahead of Time Compilation operates under the Close World Assumption, where all dependencies must be available at compile time, unlike JIT, which allows dynamic loading. In certain cases, hints about dynamic usage, such as reflection operations, class initialization, lambdas, annotations, and service loaders, may need to be provided. Configuration for this can be placed in the JAR's META-INF/native-image directory, allowing for image tuning regarding memory versus speed. However, it may still require a JVM at runtime for handling certain dynamic operations. Ultimately, AOT compilation can transform a JAR into a native executable.

3.2 Supported languages and Expanding Ecosystem

GraalVM supports a growing number of languages. This expansion brings with it a suite of new tools, including debuggers, profilers, monitors, and more. Additionally, GraalVM's flexibility extends to integration with various applications and toolkits.

The unique aspect of GraalVM is that multiple languages can coexist and execute within the same environment. Unlike traditional multi-language programs, which often run multiple languages side-by-side, GraalVM promotes seamless integration among these languages.

GraalVM's tooling supports all specific languages it hosts. Unlike tools designed for pre-compiled languages, these tools are capable of understanding and interacting with the dynamic aspects of languages like Java, Python, and more.

GraalVM's expansive ecosystem facilitates advanced integration possibilities. This means, for instance, that you can use languages like Python with MySQL instead of relying solely on SQL. This level of integration opens up new avenues for application development and interoperability across a diverse range of languages and frameworks.

3.3 Polyglot capabilities

In the context of creating a multilanguage framework, polyglot programming provides tools for seamless integration of various programming languages. Here are some key aspects:

- **Objects are Never Copied and are Converted at the Latest Possible Time**: Objects are shared and manipulated among different languages without unnecessary copying. This minimizes memory overhead and enhances performance. Data conversion into the client's physical format is deferred until the latest possible moment.
- **Availability of All Tools**: All tools and libraries are accessible to all languages. This promotes a unified and productive development experience across languages.

- **Multiple Ways to Call Foreign Languages**: The framework offers several approaches for interacting with foreign languages. They can be loaded and executed as a script, allowing dynamic interaction. They can be also compiled into classes, enabling them to be used as part of the broader application's logic. In some cases, native images can be generated from foreign language code, optimizing performance for specific use cases.

That approach allows each language to contribute its strengths to the overall application while minimizing unnecessary overhead.

Interfacing with LLVM languages often demands more boilerplate code compared to interactions within the same language. The differing memory and object models between JVM languages and LLVM languages may require additional conversion and handling, which can increase code complexity.

In some scenarios, it is more straightforward to compile JVM-based code into a native image rather than attempting to interface JVM languages directly with LLVM languages. This approach can simplify integration, enhance performance, and reduce potential complications associated with cross-language communication.

Integrating C++ code with Java can be relatively simpler when C++ code calls Java components. This is because Java has well-defined interfaces and can be loaded into the JVM environment. In contrast, invoking C++ code from Java might involve more intricate interactions due to the native nature of C++ and the complexities of JNI (Java Native Interface).

[8]

4 Where it is already useful now

The good news is that GraalVM is not just a theoretical concept; it is a practical and effective solution that delivers real-world benefits. Its versatility and capabilities make it a valuable tool in various scenarios, offering improved performance and integration options across different programming languages.

For JVM Languages:

- Simply utilizing the GraalVM JIT compiler can lead to performance improvements due to better optimization.
- Compiling code with the GraalVM compiler can result in enhanced bytecode, contributing to more efficient execution.
- Creating a Native Image using GraalVM may further boost performance, and it allows for smoother integration with other languages and components.

For Scala:

- GraalVM's JIT compiler excels at optimizing Scala code, often surpassing the capabilities of OpenJVM's JIT compiler by a factor of more than two.

For Python:

- GraalVM offers full interoperability with JVM languages, facilitating seamless interaction and data sharing.
- Python applications may experience significant speed improvements, especially when compiled to a Native Image.
- Better interoperability with C/C++ can be achieved when Python code is compiled to a Native Image.

For C/C++:

- GraalVM introduces the possibility to replace legacy C/C++ code with code written in more modern languages or integrate existing components created in such languages.
- This can be achieved through compilation into a Native Image or integration within the multi-language environment.
- Integration into frameworks and applications written in other languages (like Apache Spark[9] becomes more accessible.
- GraalVM's managed environment simplifies debugging, enhancing development workflows.
- In some cases, a performance boost can be achieved simply by recompiling code using GraalVM, without the need for extensive modifications.

One of the standout advantages of GraalVM is its ability to rewrite and optimize specific parts of a system in a more suitable language and then compile them into native executables. This flexibility allows developers to leverage the strengths of different languages within a single application, enhancing overall performance and functionality.

5 Intrinsic Limitations

While GraalVM offers remarkable capabilities, it is not without its intrinsic limitations. These limitations, though manageable, should be considered in certain scenarios:

- **Complex Configuration:** Configuring GraalVM can be a complex task, particularly when generating native images. In many cases, fine-tuning is required to achieve optimal results.
- **Java Applications' JVM Dependency:** Some Java applications may still require the Java Virtual Machine (JVM), even when compiled into a native executable. This dependency arises when applications misuse reflection and construct classes at runtime.
- **Trade-off:** Compiling Java applications into native executables provides speed benefits, particularly for small applications. However, for large and complex applications, the performance gain is not as pronounced. Java is inherently efficient for real-life applications.
- **Loss of Flexibility and Portability:** The act of compiling code into a native executable results in a trade-off. While it may boost performance, it also leads to a loss of flexibility and portability. The ability to run code on various platforms or modify it on-the-fly is restricted. By compiling into a native executable, user may loose, for example, the possibility to modify language constructs at runtime (like dynamicaly adding methods). However, GraalVM offers a possibility to profile the code and capture all actually used dynamic features at runtime and add them into the native image.
- **LLVM Language Co-existence:** Integrating LLVM-based languages such as C, C++, and Rust with JVM languages is not as straightforward as pairing two JVM languages. This challenge arises due to differences in memory and object models. Values, objects, and names must undergo conversion, and heavy communication across the LLVM-JVM boundary can potentially slow down execution. In such cases, compiling JVM languages into native images might be a more pragmatic approach.

6 External Complications

In addition to the intrinsic limitations of GraalVM, there are external complications that developers may encounter when working with a multi-language environment. These complications arise from the diverse landscape of programming languages and their specific build, deployment, and versioning systems:

- **Language-Specific Build Systems:** Different languages often come with their own elaborate build systems and makefiles, adding complexity to project management and compilation processes.
- **Language-Specific Deployment Systems:** Deploying applications written in various languages can be complicated, with each language often employing its own deployment mechanisms. These systems may also silently install dependencies.
- **Specific Bridges Between Languages:** Integrated environments with multiple languages may already make use of specific (proprietory) bridges or connectors. Internal implementations of projects might internally utilize other languages for certain tasks.
- **Mixed-Language Implementations:** Many projects, especially in the Python ecosystem, include C code alongside the primary language code.
- **Language Versions:** Managing compatibility with different language versions and dialects can be challenging. For example, the transition from Python 2 to Python 3 brought significant changes, making it difficult to support all versions simultaneously.
- **Project-Specific Environments:** Complex project-specific environments, including dependencies and configurations, further complicate multi-language development.

7 Future of Programming

The future of programming is poised for significant transformation. Programming frameworks will evolve to encompass various components, including third-party black-boxes, legacy systems, and solutions generated by artificial intelligence (AI). In many instances, the implementation language of these components may be irrelevant, or even unknown, as long as they fulfill their intended function.

This paradigm shift is already underway within the classical JVM (Java Virtual Machine) environment, where languages are chosen based on their specific strengths. For example, Scala excels in handling parallelism, while JavaScript shines in graphics-related tasks. The integration of various languages will become seamless, akin to plug-and-play functionality.

One fundamental aspect of this evolution is the clear separation of data from algorithms and logic. This demarcation is crucial and has been a long-time coming in software development. By separating these components, developers gain the flexibility to rewrite and optimize specific parts of a system in a more suitable language and then compile them into native executables. This approach enables the utilization of each language's strengths within a single application, leading to enhanced performance and functionality.

References

[1] *Apache groovy*, `https://groovy-lang.org`
[2] *Apache scala*, `https://www.scala-lang.org`
[3] *Google kotlin*, `https://kotlinlang.org`
[4] *Clojure*, `https://clojure.org`
[5] *Beanshell - lightweight scripting for java*, `https://beanshell.github.io`
[6] *Graalvm*, `http://www.graalvm.org`
[7] *Openjdk*, `https://openjdk.org`
[8] *The llvm compiler infrastructure project*, `https://llvm.org`
[9] *The apache spark*, `https://spark.apache.org`