JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Franz Miketta

Submitted at
Institute for System Software

Supervisor
tbd

Co-Supervisor
Dr. Christian Wirth

Co-Supervisor
Dr. Daniele Bonetta

March 26, 2021

CONTRIBUTING THE ECMASCRIPT PROPOSAL “MODULE BLOCKS” INTO GRAAL.JS

Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor’s Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Sworn Declaration

I hereby declare under oath that the submitted thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.

Linz, March 26, 2021

JOHANNES KEPLER
UNIVERSITY LINZ

Abstract

The thesis aims at implementing an ECMAScript proposal from the TC39 proposals, the module blocks, into the existing Graal.js interpreter. Besides the initial task of changing the parser and the runtime and unit tests, the base framework for shipping code between processes is implemented and explicit benchmarks are shown. The final product contributes to the open source version of Graal.js on the platform github.com.

Zusammenfassung

Die vorliegende Arbeit verfolgt das Ziel einen ECMAScript-Antrag aus den TC39-Anträgen, die module blocks, im vorliegenden Graal.js-Interpreter zu implementieren. Nach der Implementierung im Parser und der Runtime und entsprechenden Tests wird weiters ein Framework zum Übertragen von Code zwischen Prozessen eingerichtet und dessen Performance mit ursprünglichen Implementierungen verglichen. Die finale Implementierung wird über die Plattform github.com der Open-Source-Version von Graal.js hinzugefügt.

JOHANNES KEPLER
UNIVERSITY LINZ

Contents

List of Figures

JOHANNES KEPLER
UNIVERSITY LINZ

List of Tables

JOHANNES KEPLER
UNIVERSITY LINZ

1 Introduction

The following paragraphs serve as an introduction to the thesis by explaining the relevance both for *GraalVM* and *Graal.Js* in general and in the specific case of the ECMAScript proposal *module blocks*. The last paragraph explains the outline of the thesis.
In 2019 Oracle released the GraalVM with active development up to present days. GraalVM is a multilingual runtime with multiple core features such as certain compiler optimizations, ahead-of-time compilation, polyglot programming, LLVM runtime and the Truffle language implementation framework. With the amount of supported programming languages and the foregoing mentioned features the GraalVM can be implemented into a variety of production environments. One environment seems of particular interest: Microservices on server. The GraalVM native image was able to lower startup time and memory footprint by a significant amount as graph 1.1 shows. These savings won over the social media platform Twitter which Microservices run on GraalVM and GraalVM in return created savings for their CPU times which makes it have an environmental impact. The other core feature of GraalVM is the support of multiple languages via the Truffle framework. With this framework different languages can be implemented on top of GraalVM. The different language implementations are split up into single projects for each language. This setup leads to the project Graal.Js.
Graal.Js is the Truffle implementation of ECMAScript on the GraalVM. ECMAScript, with unofficial name JavaScript, is on spot three on the PYPL popularity index behind Java and Python indicating the programming language's relevance. From this relevance need arises to address the project's relevance. Graal.Js is the interface, on top of Truffle implemented on the GraalVM, between polyglot programs consisting of JavaScript, Java and other languages. This standalone feature lets the interpreter have a stand-out position on all the engines. Coming from other relevant engines like V8 or SpiderMonkey which cannot support polyglot programs GraalVM is on par with their feature support of ECMAScript 6 and newer standards. An important matter for relevancy of an engine is obviously the feature support. With new features being added to the standard these features have to be implemented into the engines which brings us to the new feature to be implemented module blocks.
Module blocks is an ECMAScript proposal by Daniel Ehrenberg and Surma based on inline modules. Inlining modules is a feature which is missing from the current ECMAScript. The absence has resulted into workarounds with various issues. ECMAScript cannot share code between processes thus residing on a single thread. Every module, worker and worklet needs a separate file cluttering project folders. Tasks short of stringification cannot be shared across agents. The multi-

tude of problems cited can be addressed by module blocks. Module blocks is a stage 2 proposal on ECMAScript 262 and is most likely to be implemented in the 2022 version of ECMAScript. Since it has a high relevance for polyglot programs its implementation is key to staying on top of the technology stack which brings us to the purpose of this thesis.
The main objective of this thesis is to implement the new ECMAScript stage 2 proposal module blocks into the Graal.Js engine which is implemented via the Truffle framework. Furthermore testing for this implementation will be conducted. When the general workings are set a code shipping framework and benchmarks will be included.
The thesis is divided into five further chapters. Section two explains the theoretical background of the thesis. In the following chapter three the implementation and the testing is addressed. Afterwards Section four handles benchmarking. The thesis is then rounded up in Section five and six by highlighting future work and a conclusion.

2 Background

This chapter lays out the theoretical background of this thesis by first introducing the general concepts of a high-level language cirtual machine (HLL-VM) like the Java Virtual Machine (JVM) and then going on to more specific features of the GraalVM. In the following the next abstraction layer the Truffle API is explained before coming to the explicit project Graal.Js which uses Truffle. The last section evolves around the ECMAScript parts of the topic with a brief glance at the current state of modules and what the proposal tries to achieve.

2.1 HLL-VMs

2.1.1 HLL-VM groundwork

Generally speaking a high-level language virtual machine is an abstraction layer relieving the programmer from several tasks with the main feature being platform independent development. When starting to discuss the platform independence it is foremost important to note why programs are usually platform bound.
Every computer employs some kind of an instruction set architecture (ISA) and an operating system (OS). Every developped program is bound to these two technologies. If a program is developed for a particular pair of ISA/OS it has to be ported to run on a machine with a different pair of ISA/OS. The problem arising with that is huge support overhead since now every ported version has to receive different kinds of updates. It is thus highly impractical to enforce such a development environment for every application program. By developing a high-level language virtual machine this task is posed upon the development of the VM only and all other application programs being developped don't have to focus on platform dependency resulting in a more lean development process. How is the VM doing this particular task?
High-level language virtual machines enhanced the concept of early VMs like P-code by using virtual instruction set architectures (V-ISA) encompassing code and metadata, like data structures and resource-related information, independently of platforms. The code is simpley interpreted the metadata loaded and thus turned into a machine-dependent version by the virtual machines provided emulator. This alone already stands out as a huge accomplishment but HLL-VMs come with even more features.
In today's computer landscape the highest risk comes from untrusted software run on machines.

The HLL-VM, especially the JVM, provides a metaphoric sandbox in which the untrusted application can run without making the rest of the system outside of the VM vulnerable. But although it amplifies security there are loopholes to bypass said security measures especially when untrusted software is given explicit permission to go outside of the VM provided resources. So running untrusted software is still not advised but made more secure so with a HLL-VM. As said before security is not the only feature HLL-VMs serve. Making code robust is another one.
Especially when it comes to large-scale software systems robust software is key. Here the good fit of the object-oriented model and the platform independence make HLL-VMs the top technology in the field. How much a HLL-VM supports robustness is of course depending on the used VM but on the example of the JVM strong type-checking and garbage collection which will be explained later lift a lot of beverage off the programmer's shoulder making the programmer concentrate on the mere implementation and thus making the programmer produce more robust code.
Other merits of HLL-VMs come from technologies like dynamic linking saving network bandwidth or profiling for performance.[3]
As the start of this subsection already stated a HLL-VM is an abstraction layer with a lot of features making a programmer's life considerably easier. This comes with some costs especially with performance. This cost is then highly reduced by certain techniques like the forementioned profiling. With the groundwork laid out the next section discusses certain features of the JVM.

2.1.2 The JVM

The Java Virtual Machine is as its name states developed around the language Java. Java is a general-purpose, object-oriented language with strong static types aimed as a production language. Since one language target is simplicity details of machine representation are omitted and not accessable through the language. Further safety measures have been made like auomatic storage management and checked array access. The Java sourcecode is usually compiled ahead of time (AOT) to the Java bytecode which is then run on the JVM. Ahead of time compilation means that the programmed sourcecode gets compiled into a machine specific executable form. This form is in the case of Java Java bytecode, i.e. a class-file.[1]

As stated before the JVM is the virtual sandbox environment in which an AOT-compiled java program, java bytecode, is executed. The JVM, like an actual machine, has an instruction set and access to various memory areas. With this in mind a JVM can also be directly implemented as a CPU or in various other direct ways. From knowing what the JVM is, the next paragraph dives deeper into the topic and explains how the JVM is constructed with the help of figure 2.1.
As already explained the JVM executes Java bytecode which can be in the form of *.class-files. Before the thesis dives into the structure one note has to be made: The JVM is usually delivered as a Java Runtime Environment (JRE) or Java Development Kit (JDK). These include besides the Java Virtual Machine also the Java Application Programming Interface (API) classes. These are not directly part of the JVM but play a vital role in the simplification process for programming that is

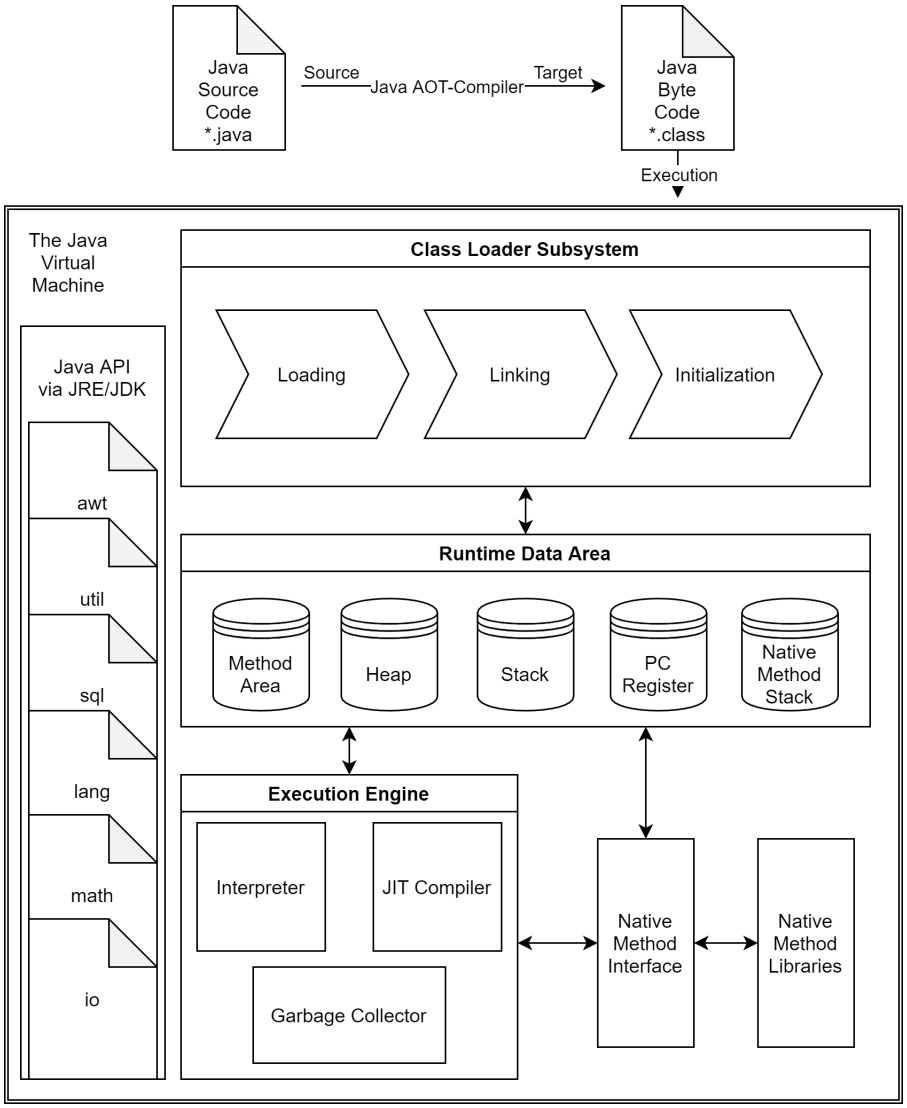

Figure 2.1: The Java Virtual Machine simplified structure

undertaken by the Java project. Coming to the execution the first substructure to mention is the Class Loader Environment.

When talking about the JVM the mentioning of classes and interfaces is inevitable. In further parts the distinction between class and interface is abbreviated as class. Multiple causes for class creation exist. The class to be created can be referenced by the constant pool of another class or a class' method can be invoked via reflection.

The creation at the start of a program works via loading the initial class and initializing it and furthermore invoking the specified method *static void main(args[])*. The complete execution is driven by this method which, as soon as the program has more than this specific method and starting class, causes loading, linking and initialization of additional classes and invoking additional methods. How do the three steps, loading, linking and initialization work? The next paragraph talks about the beginning of a class creation and loading.

When creating a class in execution the JVM transforms the implementation-independent bytecode into an implementation-specific internal representation of the class. The algorithm is executed step by step meaning that a class has to be completely loaded before linking can be started and it has to be verified and prepared to the full extant before initialization.
The loading is started by the class loader which can be the JVM bootstrap loader or a user-defined one. Before the JVM starts the class loader it checks whether the pair binary class name and class loader already exist in which case the class already exists thus eliminating the necessity of class creation. If creation proves necessary the class loader has to find the class' bytecode representation on the specific platform, usually a file with the class' name in a hierarchical filesystem. After finding, if not a specific error is thrown, the JVM parses the bytecode, which in turn might not be valid resulting in different kinds of errors. Optional superclasses may also be resolved. After these steps the pair of loader and class are saved by the JVM. Loading usually concerns the class itself however linking concerns the whole ecosystem of the respective class.
The JVM links a class by preparing the class, superclasses, element type in cas of array classes and resolving all symbolic references which as it is a recursive algorithm includes loading and linking these as well. The two strategies for resolution are an eager algorithm, meaning immediate loading and linking on verification, or loading and linking in a lazy way meaning resources are only loaded and linked when they are about to be used. At this point the JVM knows where the binary representation of a class can be found and which other classes are in the direct ecosystem of the class but the class' structure hasn't been checked yet.
Checking the class' structure is the main concern of the verification phase. In this step the class is checked for a static and structural constraints list. These include checks for appropriate type usage, number of arguments, instance initialization before usage, return types and many more. Any caught error in this step lead to ta thrown VerifyError. If no such error occurs the verification is finished and acknowledges the structual correctness of the binary class representation.
When doing the preparation of a class the JVM checks loading constraints on methods overriding those of superclasses and create and initialize static fields to default values. This phase can occur any time between creation and initializiation but imeratively before initialization.
Beforehand the resolution step with two different strategies has been discussed briefly without going into detail. This omission will be rectified now. Resolution in the context of the JVM means the process of dynamically determining conrete values from symbolic references in the run-time constant pool. The target of resolution includes classes, fields, methods, method types, method handles or dynamically computed constants. These different targets require different approaches. These usually require class loading, lookup, etc. They can also fail and throw errors or succeed resulting in a resolved dynamic binding in the constant pool. With all these preceding steps the class is now ready for initialization.
Conceptually initializing a class is fairly simple as it is the mere execution of the initialization method. This can only be invoked by certain cases or instructions like *new* or a subclass is initialized resulting in the initialization of the superclass. At this point the JVMs multithreading needs to be taken in account since synchronization is imperative. This again is solely handled by the

JVM and of no concern of the application programmer. The concrete inner workings are generally spoken handled by object states and unique initialization locks. The Initialization phase is the final stage of the class loader subsystem. Now the different data areas are explained briefly.
The method area The heap The stack The program counter (PC) register The native method stack With the dara areas explained the execution engine is the focus of the next few paragraphs.
The execution engine is mainly built up of three different components, the interpreter, the just-in-time (JIT) compiler and the garbage collector.
The interpreter is the part that executes the code directly/live including byte code getting translated to actual implementation-dependent machine code and its execution. When doing so the interpreter profiles the run code by keeping track for example of how often a certain piece of code is executed. This profiling information is then used to choose the parts of code that get jit-compiled

[2]

2.2 GraalVM

2.3 Truffle API

2.4 Graal.Js

2.5 ECMAScript modules

2.5.1 Current State

2.5.2 Proposal: Module blocks

3 Implementation

4 Evaluation

5 Future Work

6 Conclusions

Bibliography

[1] James Gosling et al. *The Java Language Specification Java SE 16 Edition*. Oracle America, 2021. URL: https://docs.oracle.com/javase/specs/jls/se16/jls16.pdf.

[2] Tim Lindholm et al. *The Java Virtual Machine Specification Java SE 16 Edition*. Oracle America, 2021. URL: https://docs.oracle.com/javase/specs/jvms/se16/jvms16.pdf.

[3] James E. Smith. *Virtual machines :: versatile platforms for systems and processes*. Amsterdam ; Boston: Morgan Kaufmann Publishers, 2005.