JKU

JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Franz Miketta

Submitted at
Institute for System Software

Supervisor
DI Raphael Mosaner

Co-Supervisor
Dr. Christian Wirth
Dr. Daniele Bonetta

April 3, 2021

CONTRIBUTING THE ECMASCRIPT PROPOSAL "MODULE BLOCKS" TO GRAAL.JS

Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

Statutory 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, April 3, 2021

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-Erweiterungsvorschlag aus den TC39-Erweiterungsvorschlägen, die module blocks, im bestehenden 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.

Contents

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 [1] 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 virtual 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 developed 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 delivering the abstraction layer?

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.[16]

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 aforementioned 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 accessible through the language. Further safety measures have been made like automatic 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.[5]

As stated before the JVM is the virtual sandbox environment in which an AOT-compiled java program, represented by the 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
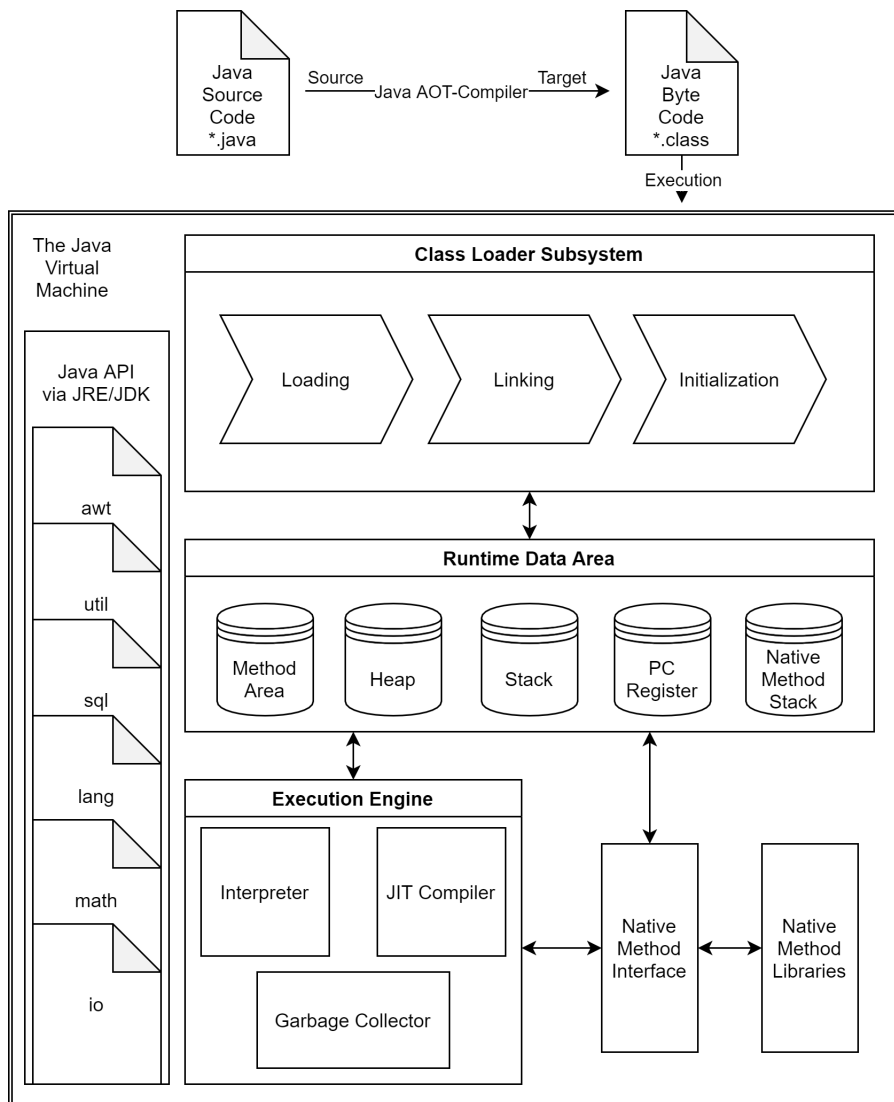
Figure 2.1: The Java Virtual Machine simplified structure

directly part of the JVM but play a vital role in the simplification process for programming that is 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 case 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 imperatively 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 JVM run-time data area is subdivided into five different data areas. These fulfill different tasks and are also different in their relation towards processes and threads as there are data areas being defined per thread and others per process. This results in some data areas existing as long as the JVM is running and some areas' existence is linked to their respective thread's existence. The data areas are, as can be seen in figure 2.1, the method area, heap, stack, program counter register and the native method stack. The next paragraphs explain which area is responsible for which task and when they are created and destroyed.

The method area is shared between all JVM threads and is created on JVM start-up. It stores all per-class structures like the constant pool or field and method data.

The heap is a JVM-defined structure, as such is created on JVM start-up, and is shared between all threads. It is an allocation space for class instances and arrays on run-time. Allocation and deallocation in the JVM is automatically handled by the garbage collector which again can be shortly defined as an automatic storage management system. The structure of the heap is not necessarily contiguous and attributes like initial, maximum and minimum size can be defined by the programmer or user.

The stack is a thread-defined structure and is itself subdivided into thread-specific stacks. It does have a fixed size which is defined at stack creation. Regarding the data a stack holds: local variables, partial results and has its share in method invocation and return. The not necessarily contiguous stack is only changed by pushing and popping frames.

The program counter (PC) register holds the current program counters of all running threads. A thread specific PC register contains the address of the thread-specific currently executed method.

The native method stack is an optional data area not supported by all JVMs. If they are supported they are conventional stacks on a thread basis. They can be of fixed or dynamic size. These stacks fulfill a similar task to the regular stacks but are for code not written in Java.

With the data 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 on the central processing unit (CPU) per instruction. This mechanism causes the interpreter to reinterpret, translate and execute the same code multiple times per instruction, code which is redundant work making the interpreter slow in comparison to AOT-compiled execution. When doing so the interpreter of the JVM profiles the run code for example by keeping track 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 hindering too many redundant interpretation steps.

A part of the byte code chosen for JIT-compilation is compiled to machine code and gets thus

turned from implementation independent form into implementation dependent form. Afterwards if this part of the program gets executed again the machine code gets executed directly out from the cache where it's stored in the meantime. This method of choosing code hotspots that get executed often for compilation rather than continuous interpretation is called adaptive compilation and is used in the Oracle Hotspot VM.

The last component left out so far is the garbage collector (GC). It takes care of objects without references resulting them to be unreachable by application code. Such an object will be removed by GC and memory will be freed up. The GC is part of inner workings and fully functions without user interaction but the programmer can make a not guaranteed call for the GC.[15]

This section explained the JVM ecosystem. It is split up between an AOT-compiler that creates Java bytecode out of Java source code. From there on the JVM executes the bytecode. Generally speaking the JVM consists of three areas, the class loader subsystem, taking care of loading, linking and initialization, the runtime data area and the execution engine, consisting of the interpreter, the JIT compiler and garbage collector. Aditionally to ease the programmer's work the JRE or JDK come with an API. What hasn't been discussed so far are technologies for making Java execute faster. These technologies are a core feature of GraalVM which is the topic of the next section.

2.2 GraalVM

As seen in the last two sections building a virtual machine for executing programming languages is a non-trivial task needing a lot of effort. A lot of this effort has to be redone for different programming languages. The core idea of GraalVM is to reduce this effort via a layered structure where each language implementation is in essence an abstract syntax tree (AST) interpreter via the Truffle API. All low level implementions like JIT compiling, data areas, memory management, optimizations and the like are reused by each language implementation connecting to the GraalVM or any JVM via Truffle. Considering performance these implementations are best run on top of GraalVM. Reasons for that are explained in the next section.

The GraalVM complete ecosystem is comprised of multiple technologies. In the runtime environment the GraalVM reuses most components from the HotSpot VM like the class loader subsystem, the runtime data area and parts of the execution engine, in particular the interpreter and the garbage collector, but replaces the HotSpot JIT compiler with the GraalVM compiler. Since the GraalVM compiler functions differently it also implements an interface between the regular HotSpot components and the GraalVM compiler. Outside of the VM it also has the Truffle framework for different language implementations. Already implemented languages are JavaScript, Ruby, R, python and LLVM-based languages like C, C++ and Rust. Thus the feature is able to end the segregation of programming languages. Another feature is its ability to be embedded in
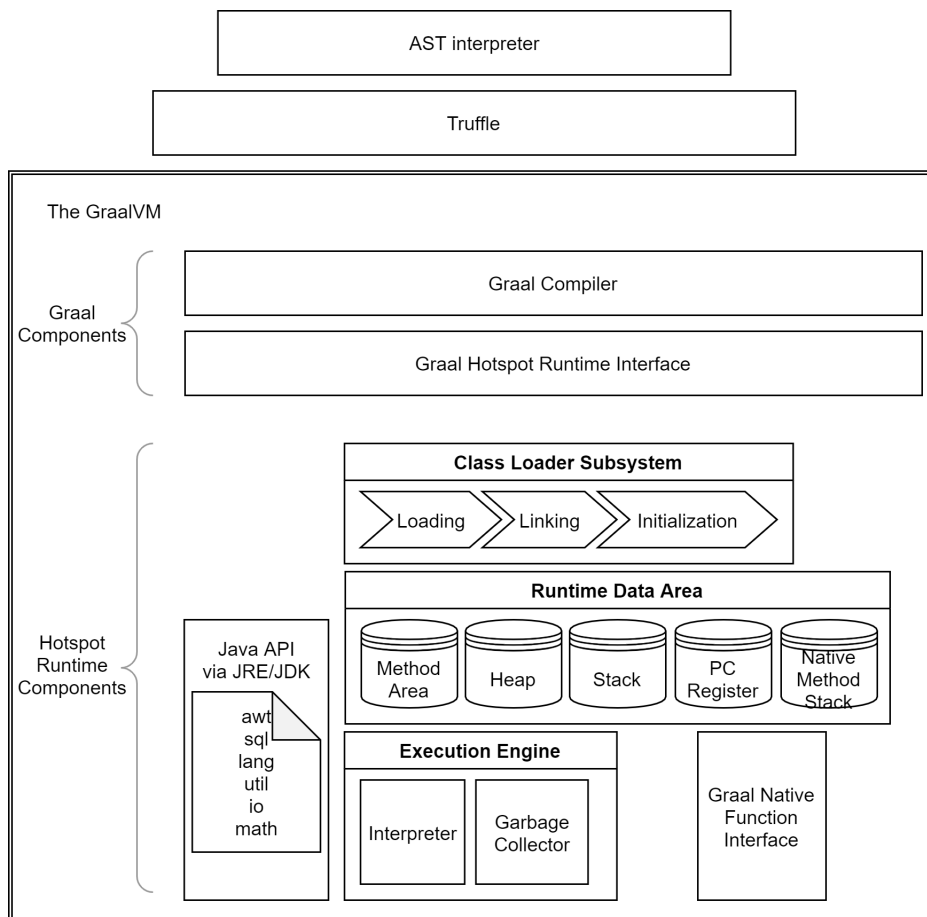
Figure 2.2: The Graal Virtual Machine simplified structure

combination with OpenJDK, node.js or inside Oracle Database, while also being able to be run standalone.

2.2.1 GraalVM Compiler

The GraalVM Compiler replaces the JIT compiler in a JRE. Other as the regular java JIT compiler it doesn't translate java bytecode directly to machine code but to the Graal IR. [2] The Graal IR is a graph-based high-level intermediate representation. This intermediate representation is key to GraalVM's aggressive optimization techniques based on optimistic assumptions such as specific types and branch prediction. Aggressive optimization also means the compilation is prone to failures which are caught by so called guards. When a guard fails execution is transferred back to interpretation. This mechanism is called deoptimization. [10] Compiled code gets saved as a VMs internal data structure and subsequent calls to the particular part of code get always directly executed in machine code omitting the unnecessary step of interpreting.

Another component the GraalVM project substitutes is the Java Native Interface [15]. It is exchanged with the Graal Native Function Interface (GNFI). [7] It allows for efficient native function calls within Java code.

2.2.2 Truffle API

The Truffle language implementation framework enables programmers to write an AST interpreter in Java for any programming language. The biggest advantage is the reuse of JVM runtime services such as automatic memory management with execution by a JVM. But how does the interpreter work?

The interpreter works by modelling language structures as nodes building the abstract syntax tree. Evaluation of the tree works by invoking the execute method which is within every type of Node. These different Node types all extend the class Node forcing them to have an execute method implementing their respective semantics. The whole tree gets then recursively evaluated by the recursive execute method calls.

Optimizations on AST level mostly work with a speculative specialization technique. The Nodes rewrite themselves into specialized variants at run time where profiling information is available. [18] As we have discussed earlier in the GraalVM compiler speculative optimizations, like a specialization, are prone to failure. Instead of going just back to interpretation a failed specialized Node rewrites itself back into a more generic version. What kind of specializations are possible via this approach?

Type specialization especially with dynamic languages is an operation reducing code on runtime and thus creating speed up. To discuss this optimization method further the add method with the + operator in the scripting language JavaScript is used. The add method in JavaScript is defined for various types including Strings and Numbers. We reduce the language's complexity to the
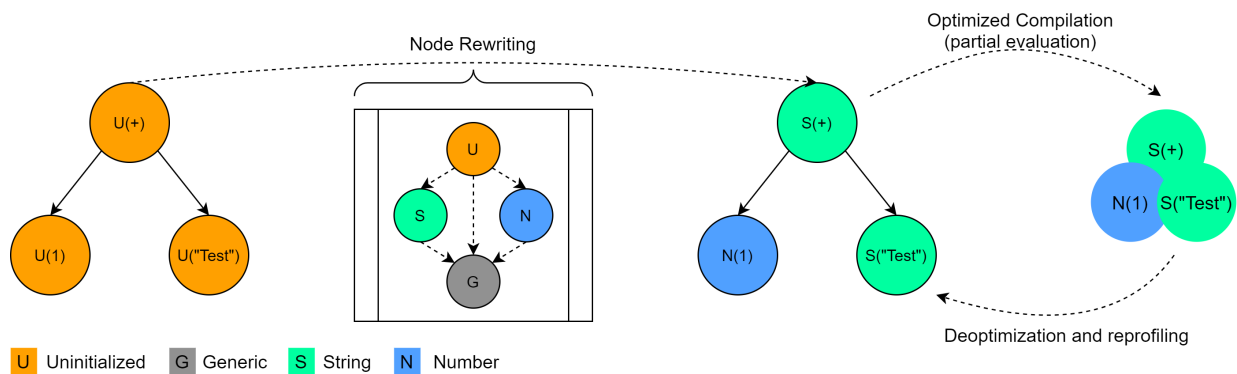
Figure 2.3: Simplified Node Rewriting example [17]

forementioned two types in the example.

```
1   var w = 1 + "Test" // results in "1Test"
2   typeof(w) // string
3   var x = 1 + 2 + "Test" // results in "3Test"
4   typeof(x) // string
5   var y = +"1" + 2 // results in 3
6   typeof(y) // number
7   var z = +"Test" + 1 // results in NaN
8   typeof(z) // number
```

Listing 2.1: JavaScript addition types minimal example

The above example code shows how addition in JavaScript works. The interpreter has to check the types of two operands from left to right before it can decide on which operation to use, an addition for numbers or one for strings.

The node rewriting process shown in figure 2.3 happens at various state to achieve different goals. In the interpreter node rewriting is used to capture profiling feedback via runtime information such as the rate of node rewrites. If certain thresholds are surpassed the specialized AST gets compiled into machine code. If the execution fails the compiled code gets deoptimized back to the AST interpreter where again nodes are rewritten and now update the existing profile information. After this step the newly specialized AST gets compiled again.

One note on the compilation process: The compiler uses as said aggressive optimization methods. This operation is successful for stable ASTs. Then compilation techniques like method inlining and eliminating interpreter dispatch code can be performed. This technique is called partial evaluation. In addition to type specialization truffle offers so called polymorphic inline caches. [11] An inline cache itself is simply caching a method from a former lookup at call site. In Truffle inline caches are built by chaining nodes which check for cached target matches which are then executed. At a predefined chain length the chain gets replaced by a polymorphic node handling all operations.

As discussed in the JVM section program loading often requires resolution. An implemented truffle language can cache resolved targets at run time by replacing an unresolved node with its re-

solved version. Thus further resolutions of this node are prevented which leads to an optimized version of the AST.

This section gave a small introduction into the Truffle API for implementing languages via an AST interpreter for running on top of a JVM and showed the optimizations that are conducted on AST level. After an AST at run time becomes stable with no more rewrites the AST is compiled dynamically to machine code. Truffle uses the JIT compiler of a JVM, in case of GraalVM the GraalVM compiler. If GraalVM is used the nodes' execute methods are inlined producing one compilation unit of the whole tree. In the next step the GraalVM compiler uses its aggressive optimization techniques for the whole inlined tree unit producing efficient machine code. These combined methods are based on the principle of the first Futamura Projection also called partial evaluation. [4] As stated in section 2.2.1 guards or also optimization points need to be implemented to check all the assumptions made before compilation. As soon as it is necessary to rewrite a node during machine code execution the control flow gets transferred back to the interpreted AST to rewrite the node either on profiling information to another specialized version or a more generic version. [10]

In conclusion the GraalVM project aims at simplifying language implementations and application programming while also trying to have a performant run time. The simplification comes from introducing a framework for implementing languages as AST interpreters which can run on top of any JVM. The performance comes especially from running the AST interpreter on top of the GraalVM which uses various techniques to bring run time performance to higher levels.

2.3 Graal.js

The Graal.js project is an AST interpreter on top of Truffle. It is in full compliance of the newest ECMAScript 2020 language specification [12, 13] and growing support of the technical committee 39 (TC39) proposals for future ECMAScript language specifications. [9, 14] The runtime can execute JavaScript and Node.js applications with the previous discussed benefits of the GraalVM technology stack. [6] As a recap the most notable ones are performance and polyglot programming support with multithreading support.

2.4 ECMAScript modules - Current state

With ECMAScript 6 the language got modules making it possible to divide code up into multiple files. Although this possibility has been around in JavaScript before, it was implemented by external libraries and not an inherent specified feature. The ability to import modules as needed came with a rise in source code size in regular JavaScript programs. When the language started out it was not unusual to have a one-line .js (JavaScript) file. As the code size rose the necessity of modules

came with it which led to a multitude of concurring solutions. As script concatenation as the basic step meant manual building and testing the method could only be used in simple projects while more difficult ones started using module loaders. Those in turn could also become complicated for larger code bases. To overcome these problems so called module bundlers, like Babel, could be used to generate the needed JavaScript code at build time. Thus the code includes all dependencies in a single concatenated file. The specification of modules in ECMAScript bundled these efforts into one concept and gave implicit module support by the language. With this specification performance of module loading was pushed towards engine implementation.
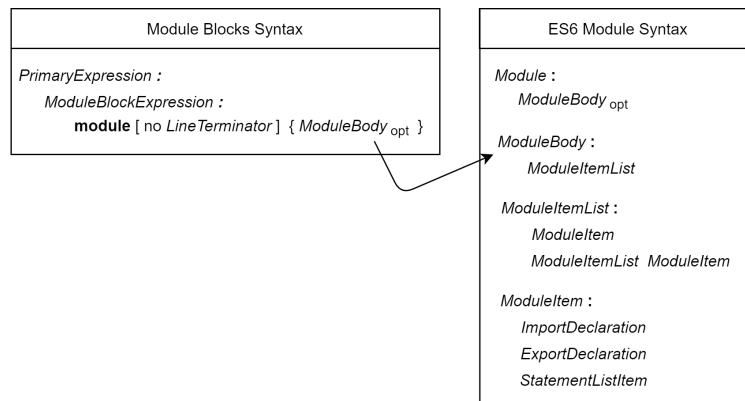
In a short note to avert term mixup. ECMAScript is a language specification for script languages. The actual code written is in JavaScript and this language is then run via engines in browsers on computers. So in hard terms an ECMAScript specification does not need to be fully supported by a particular browser and single features sometimes are left out but in general engine implementation follows the ECMAScript specification leading to the specification's features in JavaScript sourcecode.

In a broad sense modules at current state read like regular scripts with the difference of them being in strict-mode only and possible usage of imports and exports. The benefits are separate files with self-contained functionality, sharing of those files between different projects, diminishing naming conflicts, obvious naming conflicts and the robustness that comes with shared open source. But what are modules in the context of JavaScript in detail?

As a module is run in strict mode all declared variables, functions and classes are private and cannot be accessed from the outside. Only those top-level items that are explicitly exported by the keyword *export* can be used from a script importing the module. This is then done by the keyword *import* either the whole module or explicitly named items which can also be aliased.

As modules have to be loaded and are sometimes scattered throughout the web, deferring module loading is a default setting. Another note is the caching of modules. A module is only executed the first time it gets imported. All other scripts importing the module are given the already evaluated exports. This also means that a change in one of the exports is visible to all importing scripts.

This standard unfortunately left out the possibility to add inline modules into a script which results in modules just containing one exported function. This results into unnecessary files which could be avoided by inlining the module into the original script. This exact feature is discussed in the next section.

Module Blocks Syntax

PrimaryExpression :
 	
    ModuleBlockExpression :
 	
       module [no *LineTerminator*] { *ModuleBody* $_{opt}$ }

ES6 Module Syntax

Module :
    ModuleBody $_{opt}$
 	
 ModuleBody **:**
    ModuleItemList
 	
 ModuleItemList **:**
    ModuleItem
      ModuleItemList ModuleItem
 	
 ModuleItem **:**
    ImportDeclaration
    ExportDeclaration
    StatementListItem

Figure 2.4: Syntax of Module Blocks and ES6 Modules [3, 8]

2.5 Proposal: Module blocks

Module blocks are a TC39 stage 2 proposal from Daniel Ehrenberg and Surma. [3] As stated EC-MAScript does not have the feature of inlining modules. This cause several issues in programming applications, most notably code composition and file count. In the current state if code should be run in parallel an additional file is needed resulting in every Worker needing a separate file. So module blocks work like a module but are inside the script where they are also imported. One important note to take is the non closure of module blocks following the principle of coding in one place, running it anywhere. It can thus also be imported from any Realm and is code independent of a particular context. So in a sense module blocks are mainly a syntactical addition to ECMAScript as the concept is to provide a syntax for inlined module contents. These can then afterwards be imported asynchronously. Reason being the ability of module blocks to import other modules as well. As regular modules since ES6 module blocks will be cached in the module map after the first import. One note to take is, that module blocks need to be imported via dynamic import() calls rather than import statements. This is due to them not being addressable as specifier strings.

3 Implementation

4 Evaluation

5 Future Work

6 Conclusions

List of Figures

List of Tables

Franz Miketta

Bibliography

[1]   Pierre Carbonnelle. URL: `https://pypl.github.io/PYPL.html` (visited on 04/03/2021).

[2]   Gilles Duboscq et al. "Graal IR: An Extensible Declarative Intermediate Representation". In: Feb. 2013.

[3]   Daniel Ehrenberg and Surma. *JS Module Blocks*. URL: `https://github.com/tc39/proposal-js-module-blocks` (visited on 03/31/2021).

[4]   Yoshihiko Futamura. "Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler". In: *Higher-Order and Symbolic Computation* 12 (Dec. 1999), pp. 381–391. DOI: `10.1023/A:1010095604496`.

[5]   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`.

[6]   *GraalVM JavaScript Implementation*. URL: `https://www.graalvm.org/reference-manual/js/` (visited on 03/31/2021).

[7]   Matthias Grimmer et al. "An efficient native function interface for Java". In: Sept. 2013, pp. 35–44. DOI: `10.1145/2500828.2500832`.

[8]   Shu-yu Guo, Michael Ficarra, and Kevin Gibbons, eds. *ECMAScript 2022 Language Specification*. URL: `https://tc39.es/ecma262/` (visited on 03/31/2021).

[9]   Jordan Harband et al. *ECMAScript proposals*. URL: `https://github.com/tc39/proposals` (visited on 03/31/2021).

[10]   Urs Hölzle, Craig Chambers, and David Ungar. "Debugging Optimized Code with Dynamic Deoptimization." In: vol. 27. July 1992, pp. 32–43. DOI: `10.1145/143103.143114`.

[11]   Urs Hölzle, Craig Chambers, and David Ungar. "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches." In: July 1991, pp. 21–38. ISBN: 3-540-54262-0. DOI: `10.1007/BFb0057013`.

[12]   *JavaScript Compatibility*. URL: `https://www.graalvm.org/reference-manual/js%20/JavaScriptComp` (visited on 03/31/2021).

[13]   Kangax, Webbedspace, and Denis Pushkarev. *ECMAScript compatibility table*. URL: `kangax.github.io/compat-table/es2016plus` (visited on 03/31/2021).

[14]   Kangax, Webbedspace, and Denis Pushkarev. *ECMAScript compatibility table*. URL: `kangax.github.io/compat-table/esnext` (visited on 03/31/2021).

[15] 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`.

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

[17] Thomas Würthinger et al. "One VM to Rule Them All". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 187–204. ISBN: 9781450324724. DOI: `10.1145/2509578.2509581`. URL: `https://doi.org/10.1145/2509578.2509581`.

[18] Thomas Würthinger et al. "Self-optimizing AST interpreters". In: *ACM SIGPLAN Notices* 48 (Jan. 2013), pp. 73–82. DOI: `10.1145/2480360.2384587`.