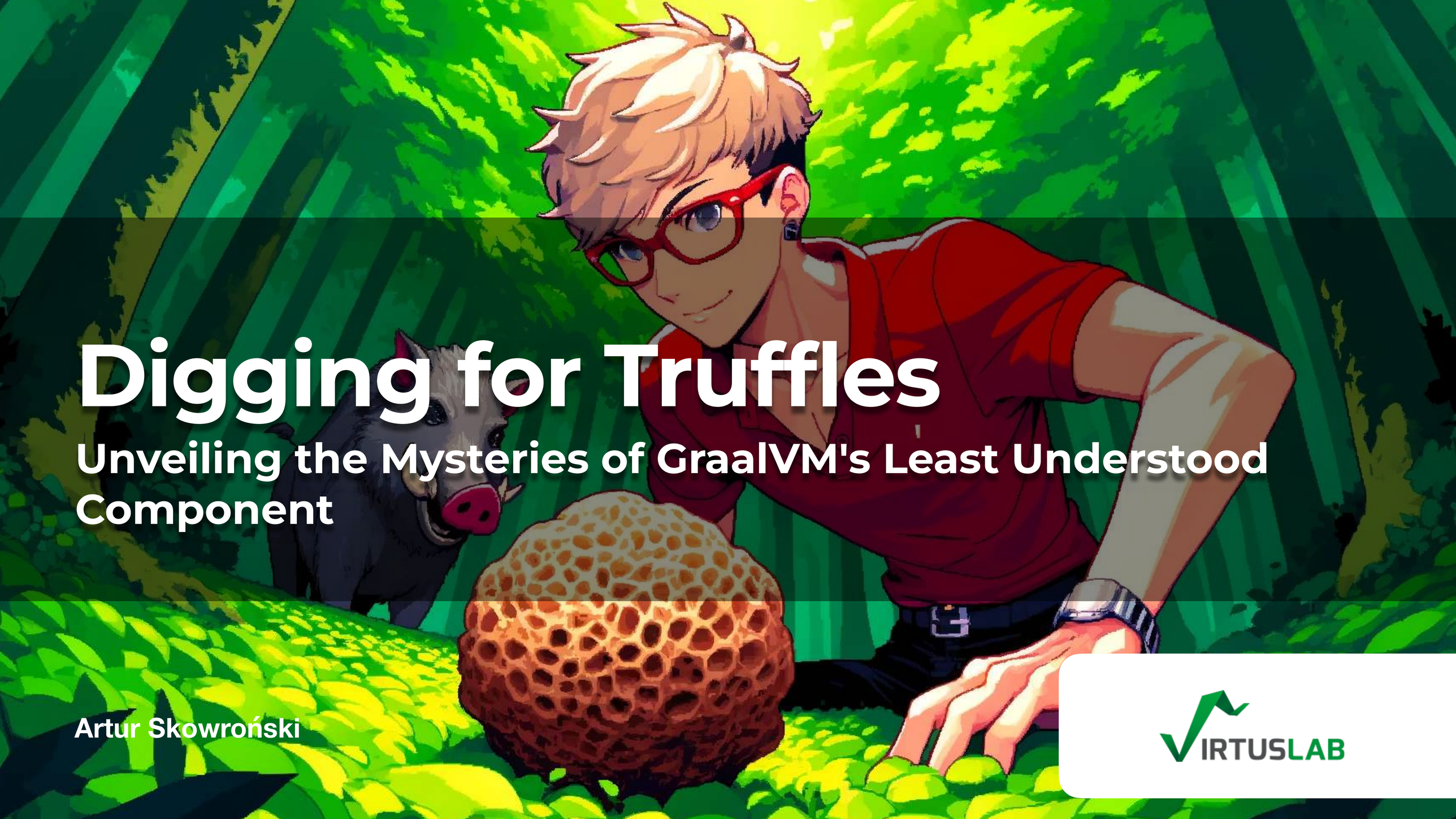


Digging for Truffles

Unveiling the Mysteries of GraalVM's Least Understood Component

Artur Skowroński

VIRTUSLAB

Join at

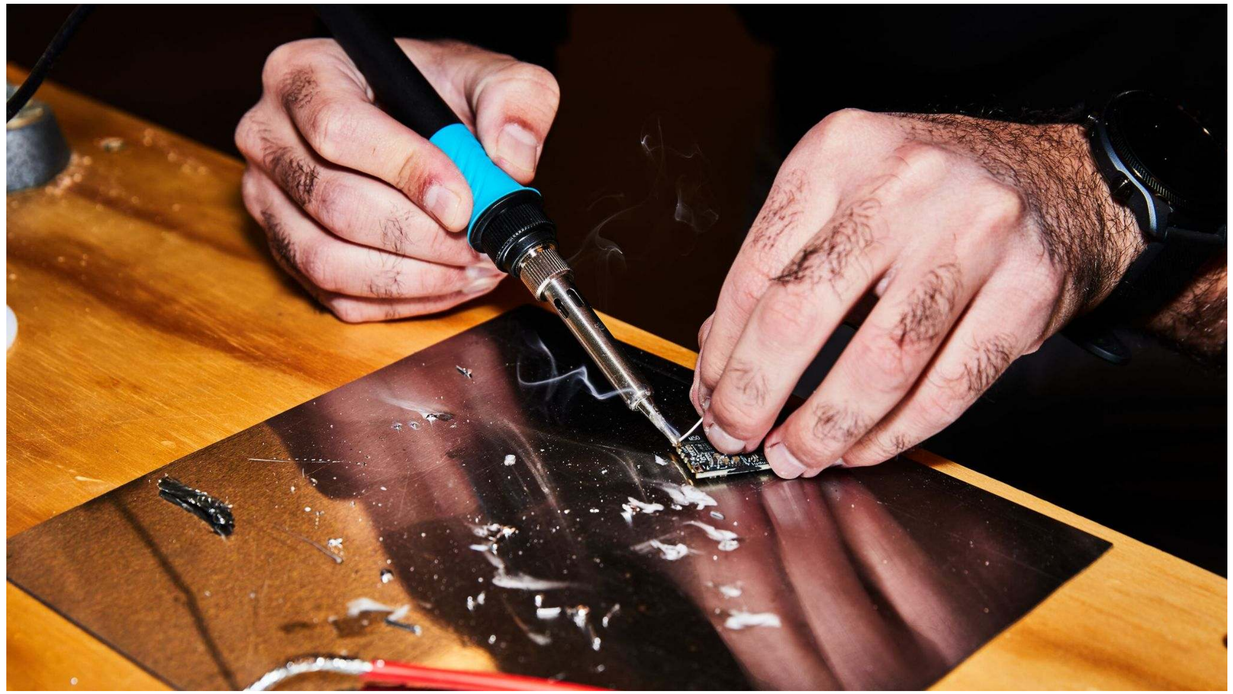
slido.com

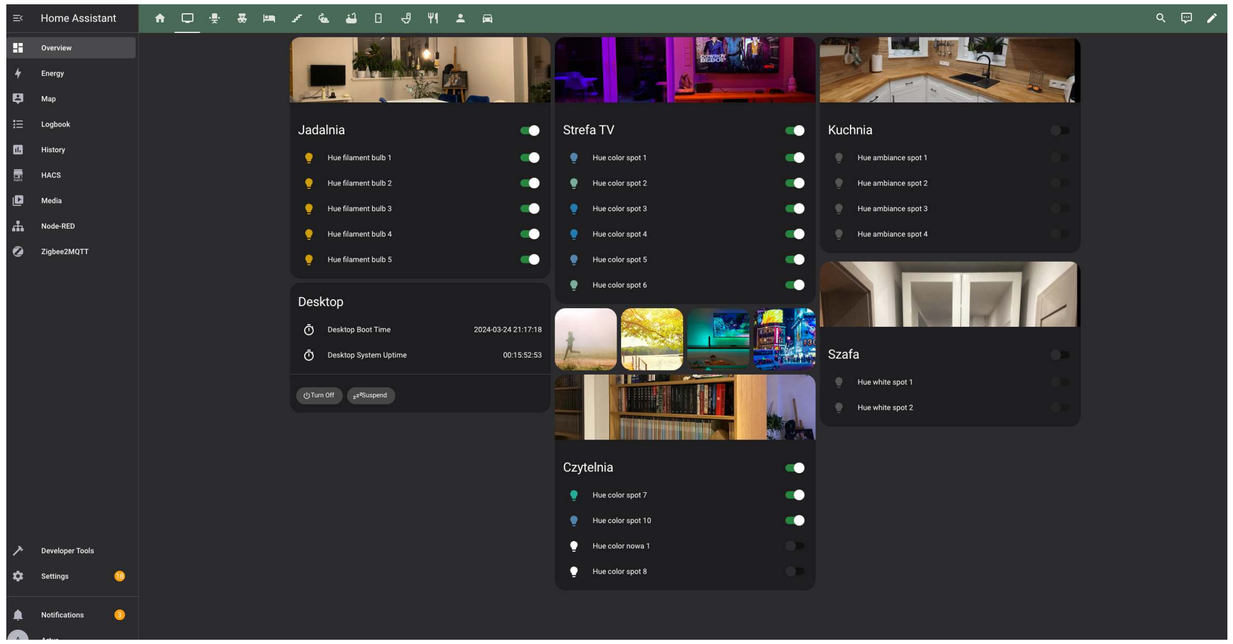
#geecon

Artur Skowronski

Head of Java / Kotlin Development

VIRTUSLAB




Home Assistant
Overview
Energy
Map
Logbook
History
HACS
Media
Node-RED
Zigbee2MQTT
Developer Tools
Settings
Notifications
Jadalnia
Hue filament bulb 1
Hue filament bulb 2
Hue filament bulb 3
Hue filament bulb 4
Hue filament bulb 5
Desktop
Desktop Boot Time
2024-03-24 21:17:18
Desktop System Uptime
00:15:52:53
Turn Off
Suspend
Strefa TV
Hue color spot 1
Hue color spot 2
Hue color spot 3
Hue color spot 4
Hue color spot 5
Hue color spot 6
Czytelnia
Hue color spot 7
Hue color spot 10
Hue color nowa 1
Hue color spot 8
Kuchnia
Hue ambiance spot 1
Hue ambiance spot 2
Hue ambiance spot 3
Hue ambiance spot 4
Szafa
Hue white spot 1
Hue white spot 2

Szukaj w Google

Szczęśliwy traf

Mój profil Moja biblioteka

Google Scholar

Dowolny język Tylko język polski

Stań na ramionach olbrzymów

One VM to Rule Them All

Thomas Würthinger* Christian Wimmer* Andreas Wöß† Lukas Stadler†
Gilles Duboscq† Christian Humer† Gregor Richards§ Doug Simon* Mario Wolczko*

*Oracle Labs †Institute for System Software, Johannes Kepler University Linz, Austria §S³ Lab, Purdue University
{thomas.wuerthinger, christian.wimmer, doug.simon, mario.wolczko}@oracle.com
{woess, stadler, duboscq, christian.humer}@ssw.jku.at gr@purdue.edu

Abstract

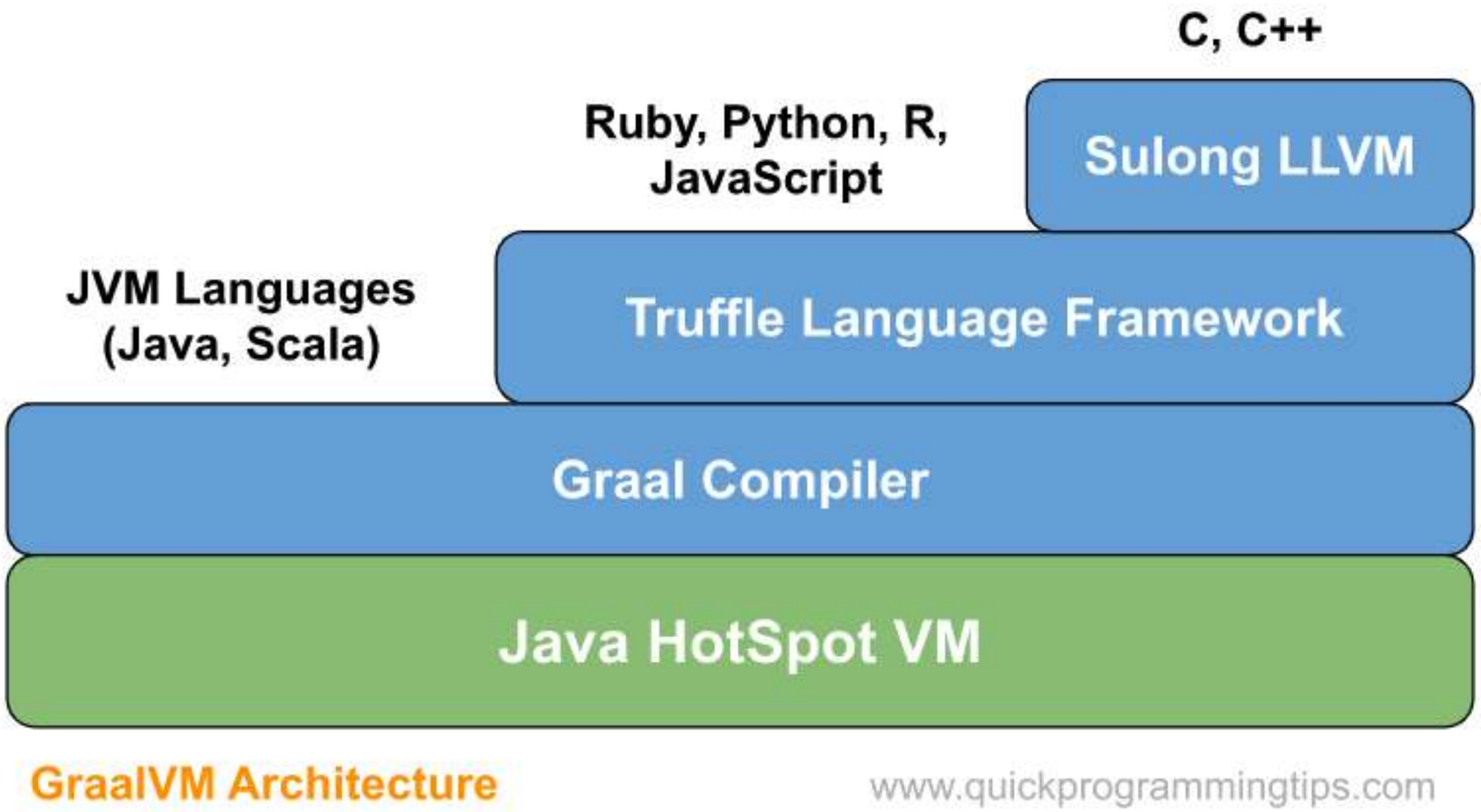
Building high-performance virtual machines is a complex and expensive undertaking; many popular languages still have low-performance implementations. We describe a new approach to virtual machine (VM) construction that amortizes much of the effort in initial construction by allowing new languages to be implemented with modest additional effort. The approach relies on abstract syntax tree (AST) interpretation where a node can rewrite itself to a more specialized or more general node, together with an optimizing compiler that exploits the structure of the interpreter. The compiler uses speculative assumptions and deoptimization in order to produce efficient machine code. Our initial experience suggests that high performance is attainable while preserving a modular and layered architecture, and that new high-

These implementations can be characterized in the following way:

- Their performance on typical applications is within a small multiple (1-3x) of the best statically compiled code for most equivalent programs written in an unsafe language such as C.
- They are usually written in an unsafe, systems programming language (C or C++).
- Their implementation is highly complex.
- They implement a single language, or provide a bytecode interface that advantages a narrow set of languages to the detriment of other languages.

In contrast, there are numerous languages that are popu-

Onward! 2013, October 29–31, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2472-4/13/10/13/10...$15.00.
http://dx.doi.org/10.1145/2509578.2509581


C, C++
Ruby, Python, R, JavaScript
Sulong LLVM
JVM Languages (Java, Scala)
Truffle Language Framework
Graal Compiler
Java HotSpot VM
GraalVM Architecture
www.quickprogrammingtips.com

GraalVM

Premium PL

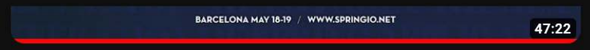
graalvm

Główna
Shorts
Subskrypcje
YouTube Music

Ty
Twój kanał
Historia
Twoje filmy
Do obejrzenia
Pobrane
Twoje klipy
GraalVM Native Im...
Pokaż więcej

Subskrypcje
Michał Pisarski T...
Little Angel: Nurs...
Podsluchane.pl
Napisy Końcowe
GiantGrantGames
VaatiVidya
Okiem Deva
Pokaż jeszcze 194

Odkrywaj
Na czasie
Muzyka
Filmy



GraalVM Native Image — Faster, Smarter, Leaner

12 tys. wyświetleń • 11 miesięcy temu

Java

With support from all the leading microservice frameworks, compatible SDKs from all of the top cloud platforms, and ease of ...

Introduction | GraalVM | Native Image | Startup Time | Demo | Memory Consumption | JIT... Rozdziały: 14

[Bydgoszcz JUG #28] Szymon Stępniak - GraalVM i generowanie natywnych obrazów, czyli Java bez JVM

1,4 tys. wyświetleń • 4 lata temu

Bydgoszcz JUG

Programiści Java przyzwyczaili się do wirtualnej maszyny Java, kompilatora JIT, konieczności "rozgrzewania" aplikacji w celu ...

GraalVM Native Image: Hello World

22 tys. wyświetleń • 2 lata temu

GraalVM 🚀

A quick introduction to the basics of **GraalVM** Native Image ahead-of-time compilation.

GraalVM & Spring Boot: Building a Native Executable | Marco Reacts

5,6 tys. wyświetleń • 4 miesiące temu

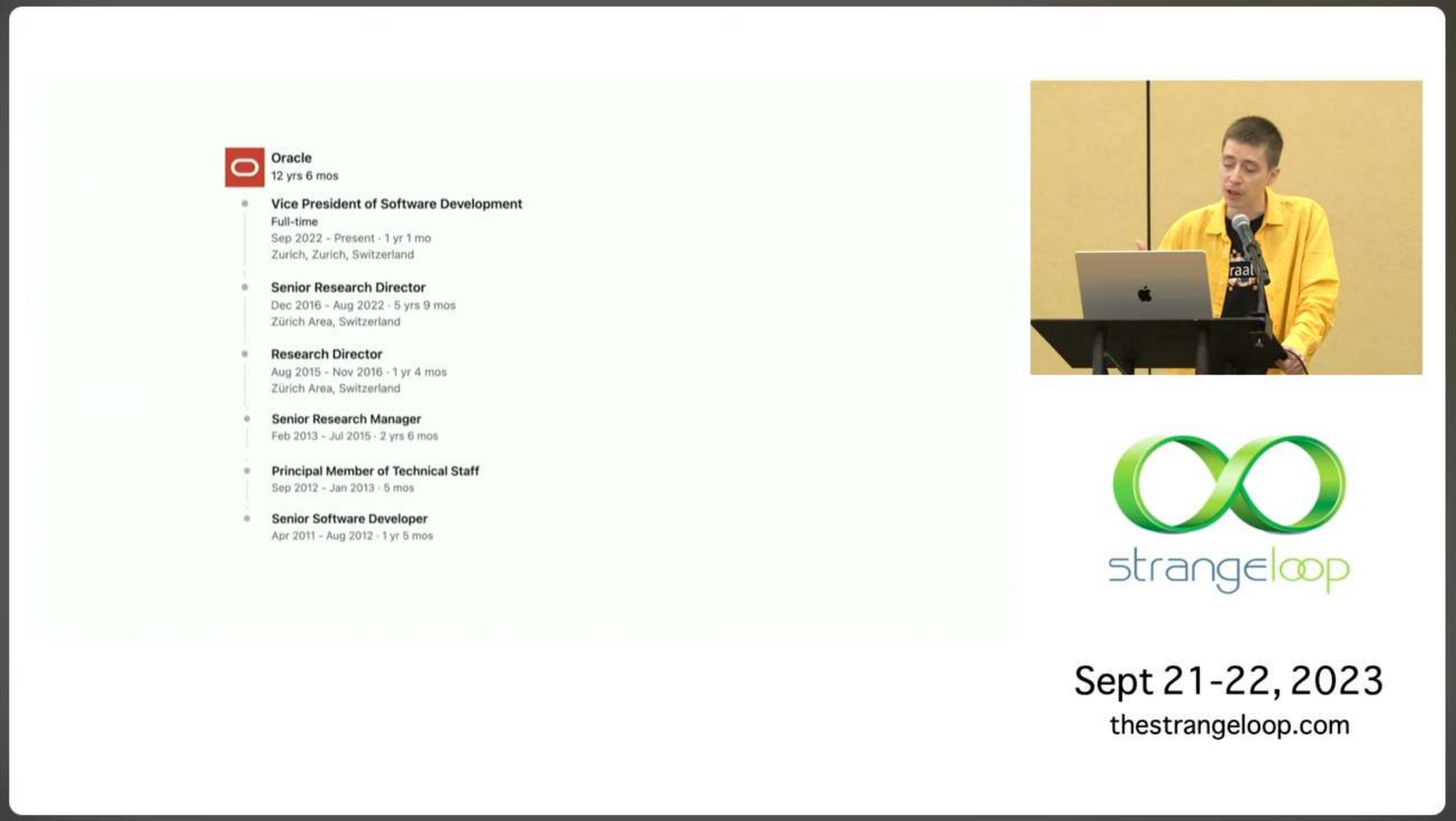
Marco Codes

You heard about **GraalVM**? You want to turn your Java webapp into a native executable to benefit from massive startup time ...

Intro | The App We Are Going To Convert | Prerequisites (SDKMan, GraalVM, WSL, Gradle Plugin) |... Rozdziały: 7

Premium PL

Szukaj



"Turning GraalVM from Research to Product" by Thomas Wuerthinger (Strange Loop 2023)

Strange Loop Conference
80,7 tys. subskrybentów
Subskrybuj

Podoba mi się | Udostępnij | Pobierz | Klip

3 tys. wyświetleń 6 miesięcy temu ST. LOUIS

GraalVM is a new runtime system that provides instant startup and low footprint for Java applications via ahead-of-time compilation. It also has polyglot capabilities that allows to execute and interoperate with many languages including JavaScript, Python, Ruby, WebAssembly, and native languages. This session discusses lessons learned while turning the vague research ideas behind GraalVM into a released product over the timespan of more than a decade, how we experienced our entry to the developer community, and also major technical questions where we now think differently. ...więcej

4 komentarze Sortuj według

Wszystkie | Źródło: Strange Loop Confere... | P

"The Economics of Programming Languages" by...
Strange Loop Conference
116 tys. wyświetleń • 6 miesięcy...
43:58

Shorts

Finally! GraalVM and Java 21...
3,3 tys. wyświetleń

What is a ConcurrentMo...
6,7 tys. wyświetleń

How can you invoke a metho...
7,3 tys. wyświetleń

"Monad I Love You Now Get Out Of My Type System" by Gjeta...
Strange Loop Conference
27 tys. wyświetleń • 1 rok temu
44:45

Maximizing Performance with GraalVM
InfoQ
2,1 tys. wyświetleń • 4 lata temu
49:31

Documentation Office Hours 5th April 2024
Ubuntu OnAir
20 wyświetleń • 3 godziny temu
Nowy
23:21

Programming language talks from Strange Loop
Strange Loop Conference · Playlista
72 filmy

50 years of Prolog: Becoming

SAY NATIVE IMAGE
ONE MORE TIME
imgflip.com

Unleash the power of your applications with Micronaut and GraalVM

Session type: ARCHITECTURE

In this talk, Micronaut committer Álvaro Sánchez-Mariscal, will demonstrate how you can quickly build optimised Microservices with Micronaut & GraalVM Native Image. Attendees will learn how the combination of GraalVM Native Image and Micronaut can lead to efficient, highly performant, and optimised applications that can be perfectly deployed to environments like Kubernetes or serverless platforms. There will be a live coding demo of an application using Micronaut Data and GraalVM.

Álvaro Sánchez-Mariscal Arnaiz

Álvaro is a passionate developer and agile enthusiast with over 22 years of experience. He is now a Principal Member of Technical Staff at Oracle Labs, where he is a Micronaut committer, helping to maintain and evolve the open-source framework. Before that, he was a Staff Engineer at VMWare, where he led the creation of new products to build, package, verify and publish software solutions with confidence in the VMware Marketplace. Previously, he spent some years working in different industries like gambling games (Odobo) and fintech (4Finance). Prior to that, he created his own company, Salenda, in 2005, a software factory and Atlassian Solution Partner headquartered in Madrid, Spain. Adaptavist acquired Salenda in 2019. Previously, he worked at companies like IBM BCS, Sun Microsystems or BEA Systems, where he was recognised as BEA Technical Director, an MVP awards program. He was also one of the initial founders and member of the Board of Directors of javaHispano in 2002, the world's largest Spanish-speaking Java User Group. Álvaro has spoken in 20 different countries at conferences like Devoxx BE/FR/MA, several Voxxed Days, GeeCON, JavaLand, JavaZone, jPrime, Codemotion and Commit Conf, among others. In his spare time, as well as coding and experimenting with new technologies, he likes to spend time with his wife and children, support CD Leganés football team, and play paddle tennis.

Going AOT: Everything you need to know about GraalVM for Java

Session type: ARCHITECTURE

In the past few years GraalVM got widely adopted by the Java ecosystem, and now it's hard to imagine starting a new project, especially for the cloud, without at least considering going native. However, taking a new technology to production requires research and preparation. What is the best way to build and deploy such native executables? Once deployed, how can I monitor them? Can I test them as I do with regular Java applications? What if startup is less important, how do I optimize for peak performance and latency? How do I use frameworks and libraries? This session will cover all of those steps and illustrate them with examples. We'll also have plenty of time for questions – bring them all!

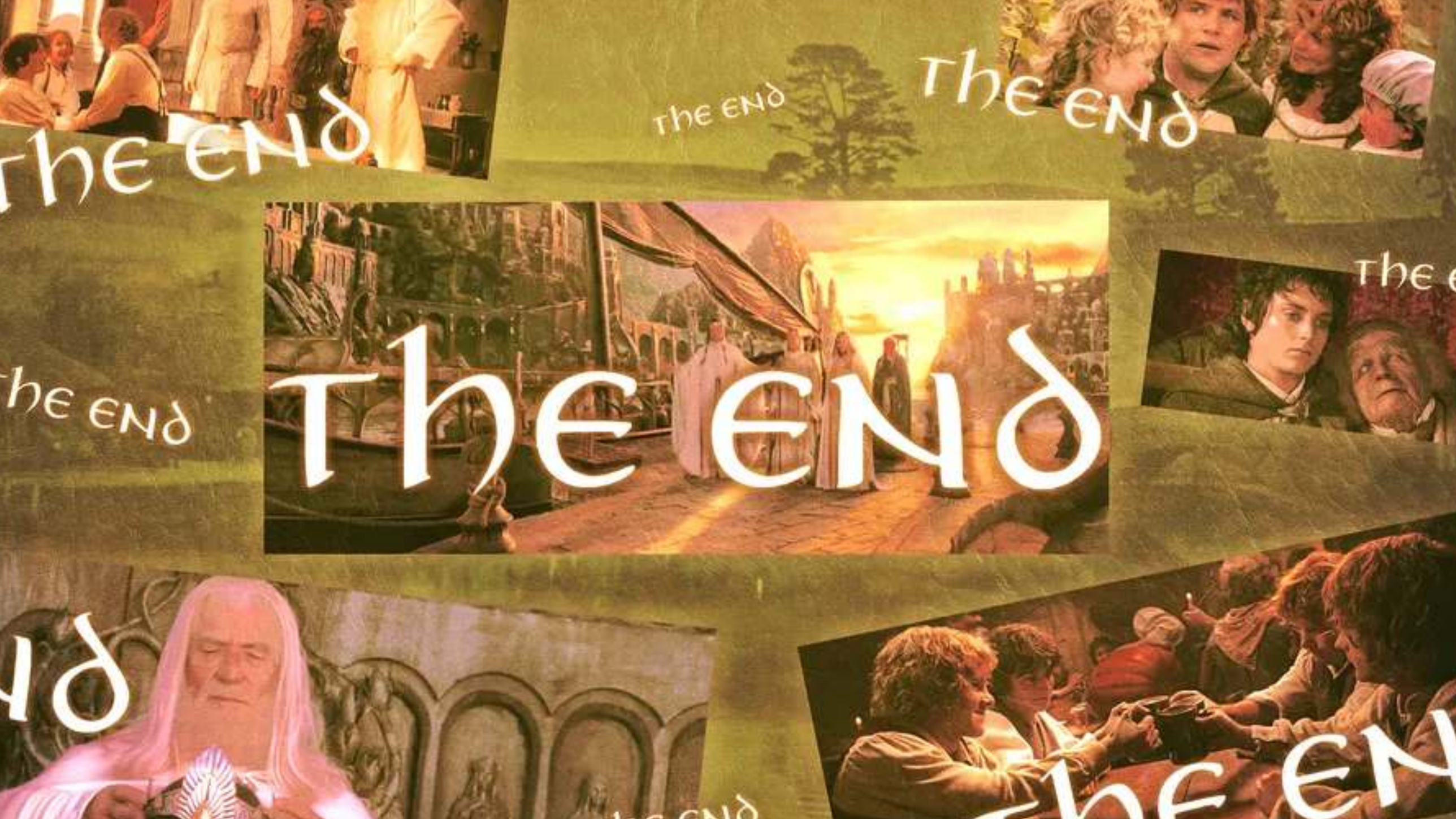
Alina Yurenko

Alina is a developer advocate for GraalVM at Oracle Labs, a research & development organization at Oracle. Loves both programming & natural languages, compilers, and open source.

Truffle Language Implementation Framework

The Truffle language implementation framework (Truffle) is an open source library for building tools and programming languages implementations as interpreters for self-modifying Abstract Syntax Trees. Together with the open source Graal compiler, Truffle represents a significant step forward in programming language implementation technology in the current era of dynamic languages.

That's all Folks!


The End
The End
The End
The End
The End
The End
The End

How to draw an owl

1.

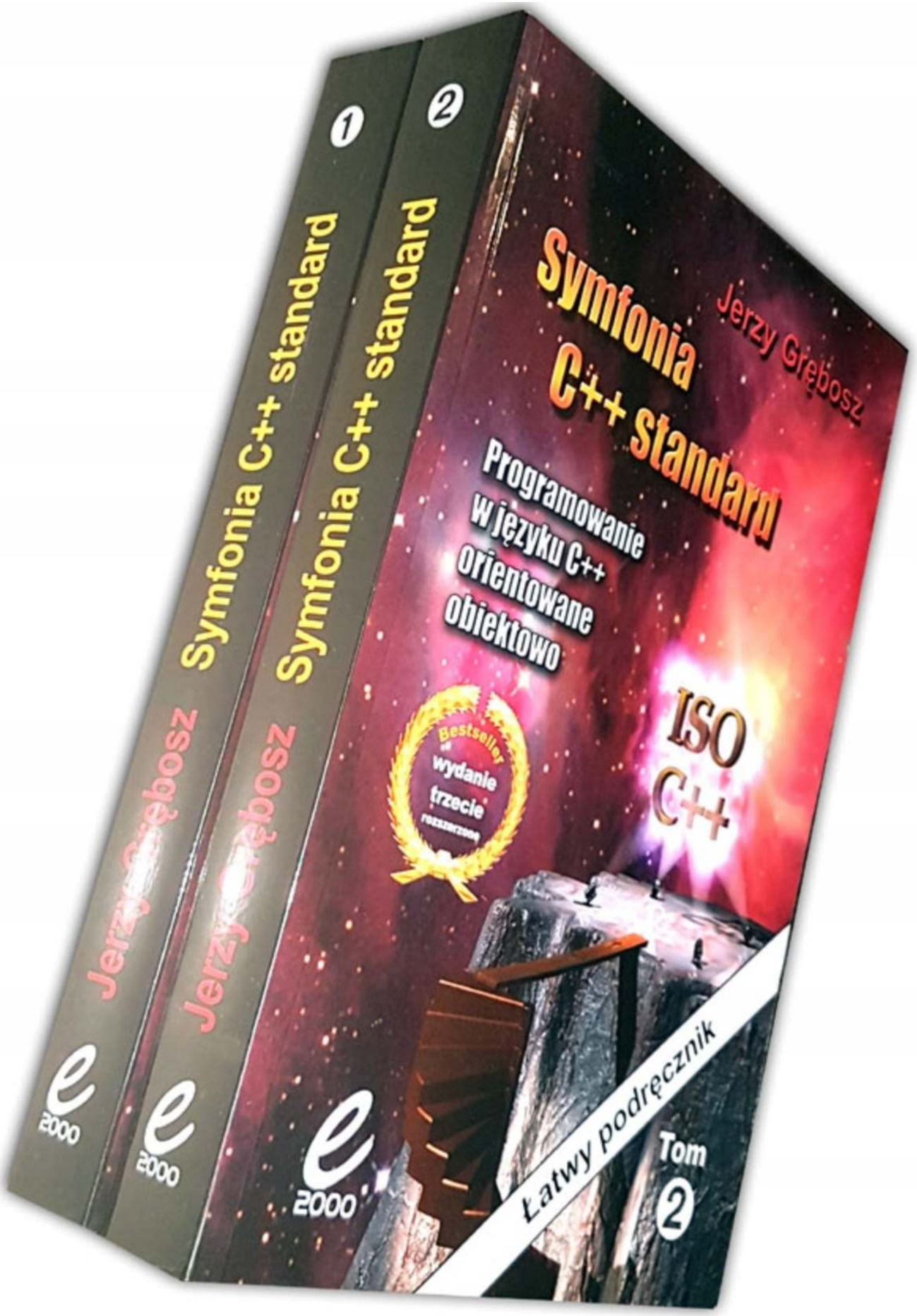
2.

1. Draw some circles

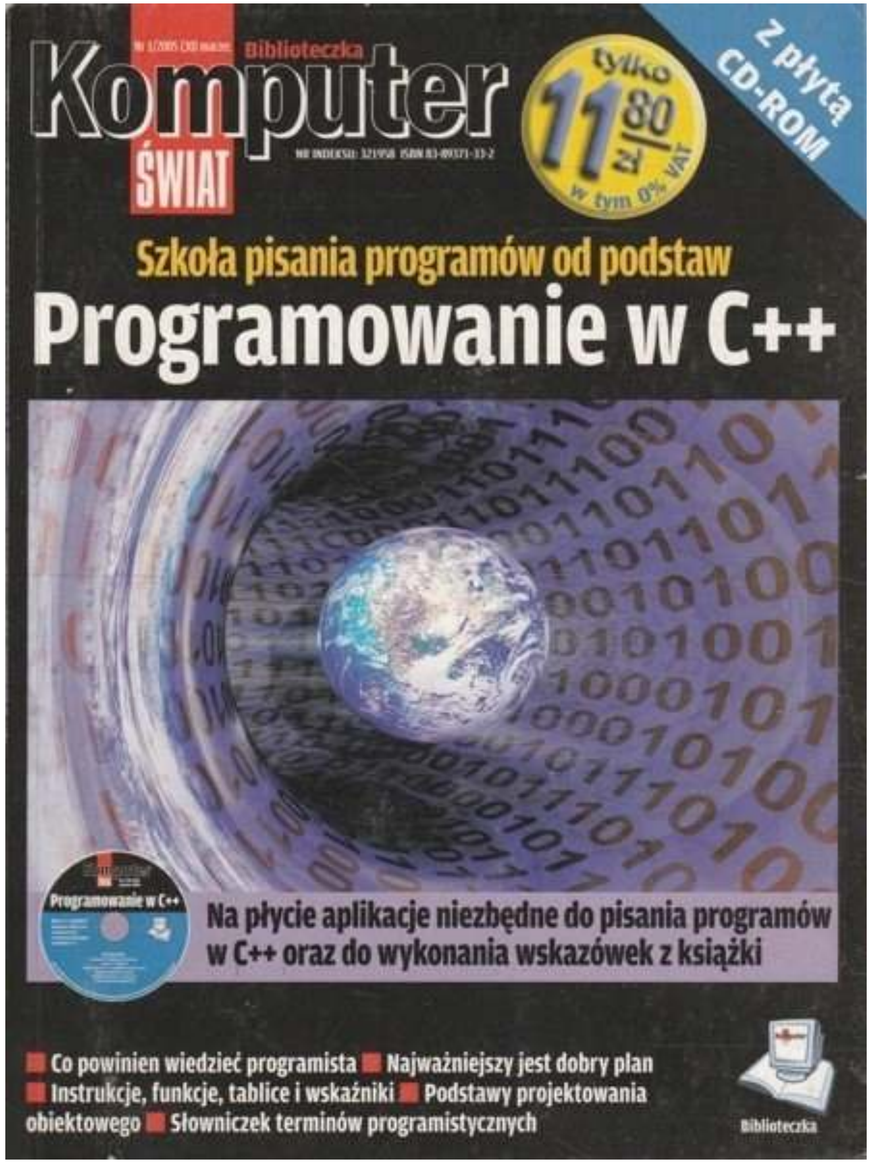
2. Draw the rest of the fucking owl

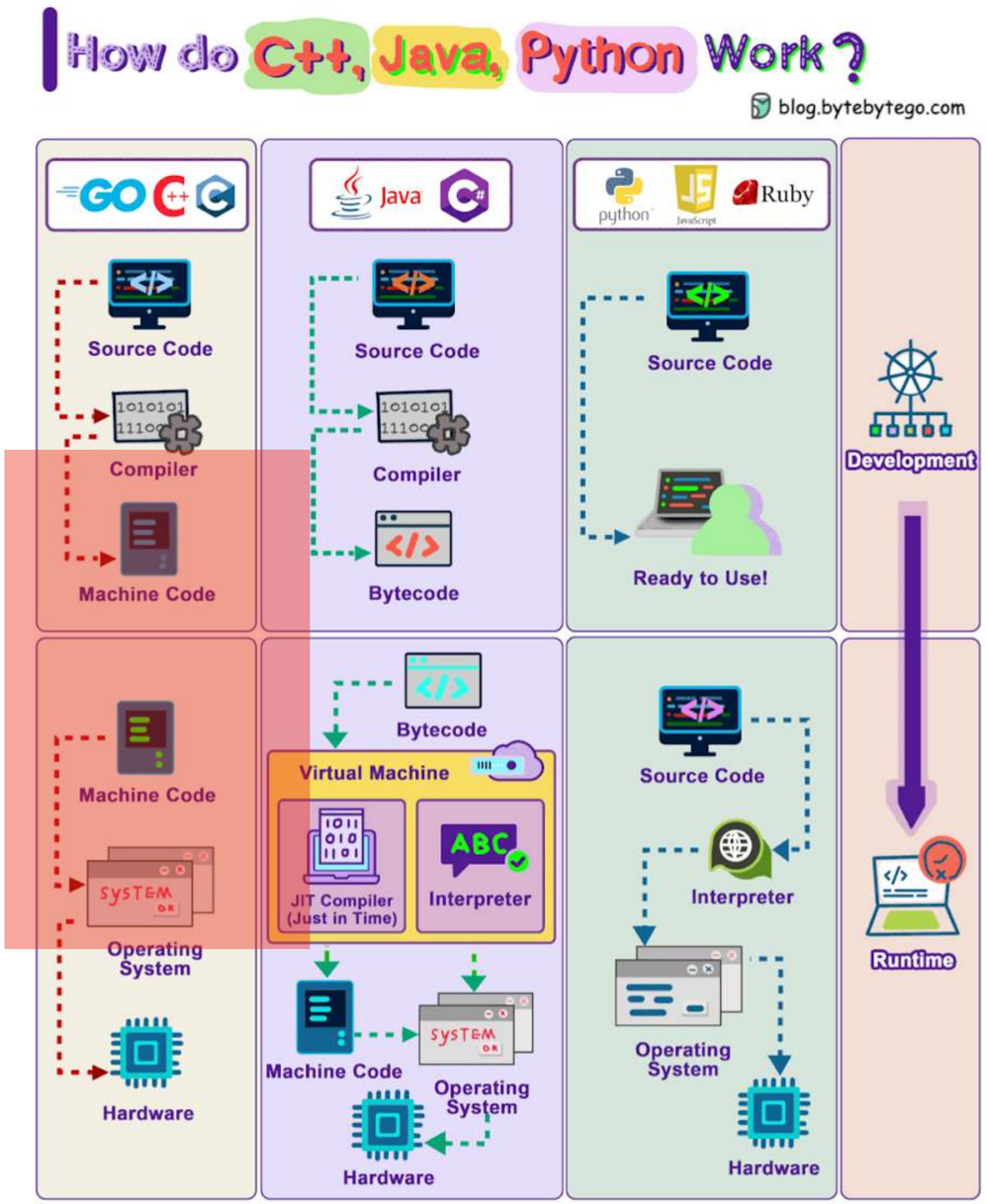
155 Slides / 45? minutes

Let's start from the beginning

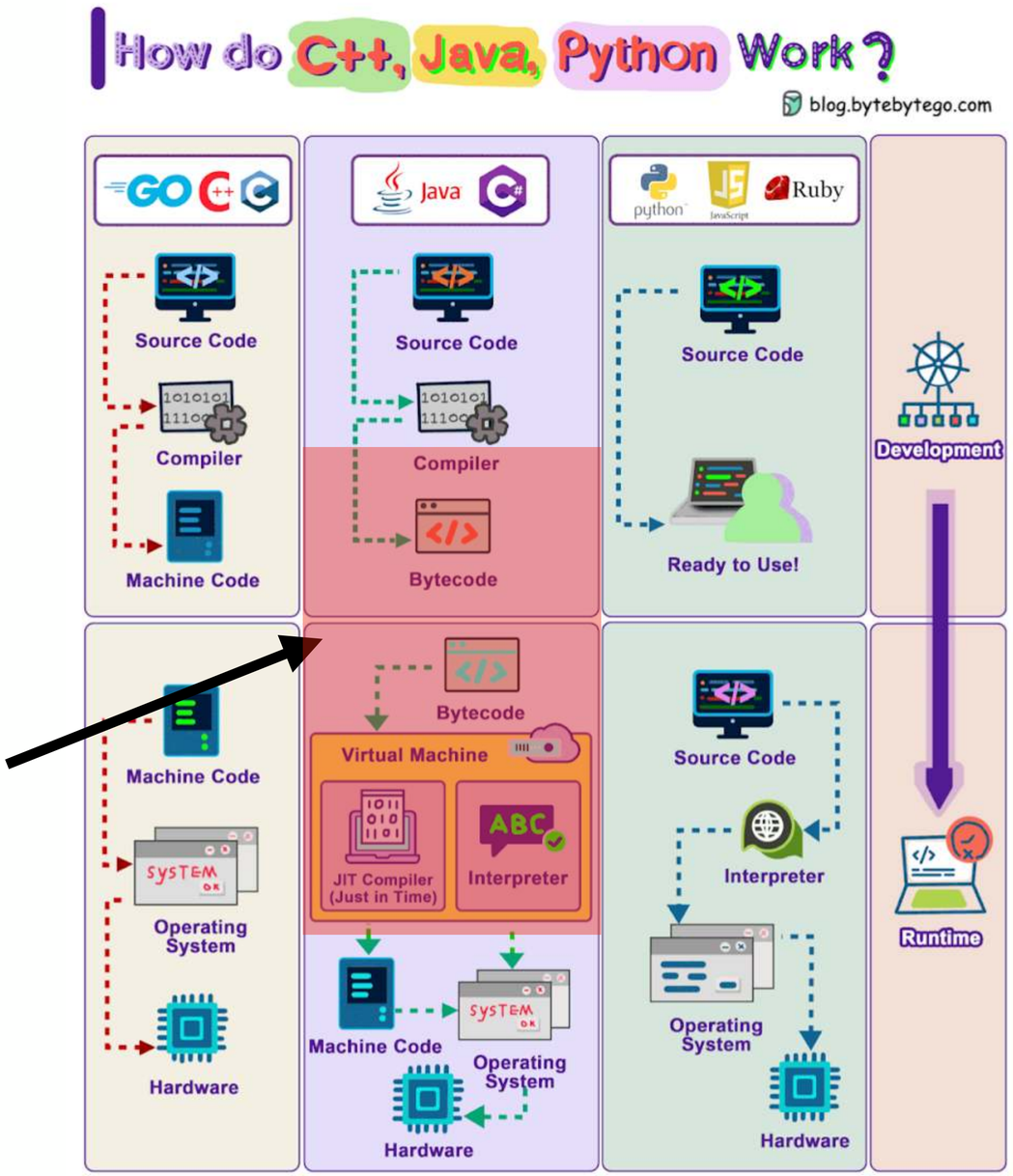

1
Symfonia C++ standard
Jerzy Grębosz
2000
2
Symfonia C++ standard
Jerzy Grębosz
2000
Symfonia
C++ standard
Jerzy Grębosz
Programowanie
w języku C++
orientowane
obiektowo
Bestseller
wydanie
trzecie
rozszerzone
ISO
C++
2000
Łatwy podręcznik
Tom
2

Discovery
Family

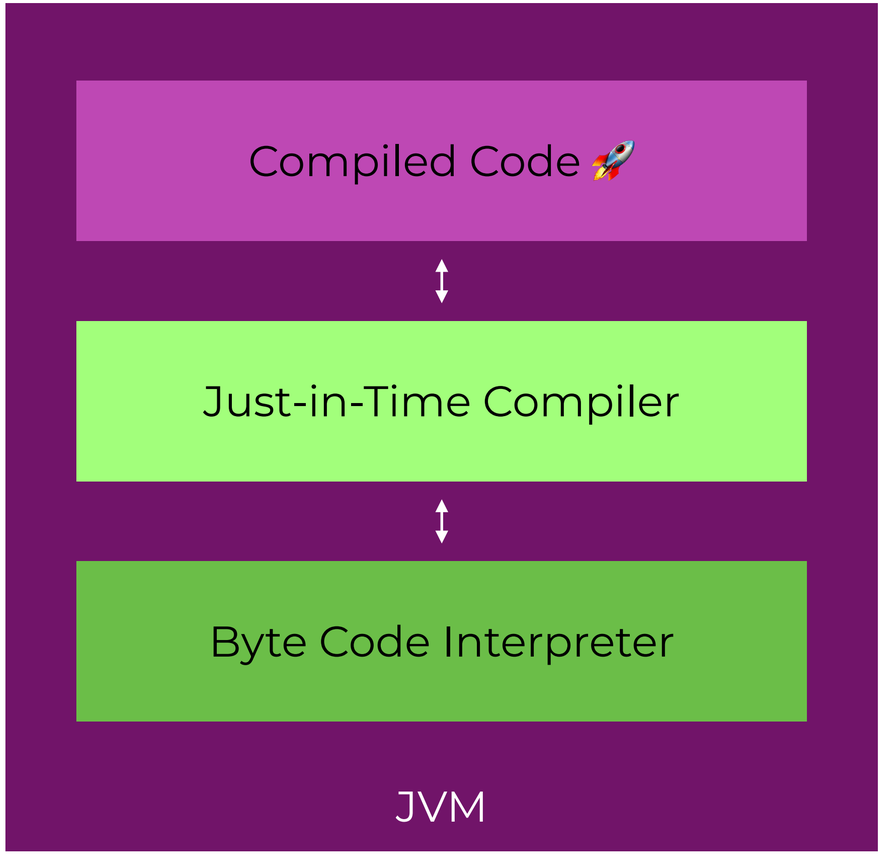

Biblioteczka
Komputer
ŚWIAT
tylko 11,80 zł w tym 0% VAT
Z płytą CD-ROM
Szkoła pisania programów od podstaw
Programowanie w C++
Na płycie aplikacje niezbędne do pisania programów w C++ oraz do wykonania wskazówek z książki
Co powinien wiedzieć programista
Najważniejszy jest dobry plan
Instrukcje, funkcje, tablice i wskaźniki
Podstawy projektowania obiektowego
Słowniczek terminów programistycznych
Biblioteczka


How do C++, Java, Python Work?
blog.bytebytego.com
GO
C++
Source Code
Compiler
Machine Code
Machine Code
Operating System
Hardware
Java
Source Code
Compiler
Bytecode
Bytecode
Virtual Machine
JIT Compiler (Just in Time)
Interpreter
Machine Code
Operating System
Hardware
python
JavaScript
Ruby
Source Code
Ready to Use!
Source Code
Interpreter
Operating System
Hardware
Development
Runtime

COMPILATION IS COMPILATION
imgflip.com


How do C++, Java, Python Work?
blog.bytebytego.com
GO
C++
Source Code
Compiler
Machine Code
Machine Code
Operating System
Hardware
Java
Source Code
Compiler
Bytecode
Bytecode
Virtual Machine
JIT Compiler (Just in Time)
Interpreter
Machine Code
Operating System
Hardware
python
JavaScript
Ruby
Source Code
Ready to Use!
Source Code
Interpreter
Operating System
Hardware
Development
Runtime

AND THEN WE TOLD THEM
WRITE ONCE RUN ANYWHERE
memegenerator.net


Compiled Code 🚀
Just-in-Time Compiler
Byte Code Interpreter
JVM

GraalVM

```
public static int add(int a, int b) {

    return a + b;

}
```

javac

```
0: iload_1

1: iload_2

2: iadd

3: ireturn
```

ByteCode

```
0: iload_1

1: iload_2

2: iadd

3: ireturn
```

ByteCode

```
0: iload_1
1: iload_2
2: iadd
3: ireturn
```

ByteCode

```
0: iload_1
1: iload_2
2: iadd
3: ireturn
```

ByteCode

```
0: iload_1
1: iload_2
2: iadd
3: ireturn
```

ByteCode

```
public static int add(int a, int b) {
    return a + b;
}
```

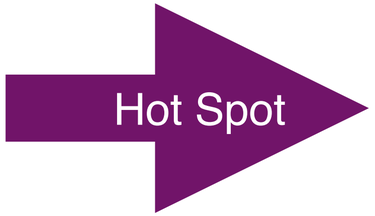
```
public static int add(int a, int b) {
    return a + b;
}
```

```
public static int add(int a, int b) {
    return a + b;
}
```

```
public static int add(int a, int b) {
    return a + b;
}
```

```
public static int add(int a, int b) {
    return a + b;
}
```

HotSpot Analysis




```
0: iload_1

1: iload_2

2: iadd

3: ireturn
```

HotSpot Analysis

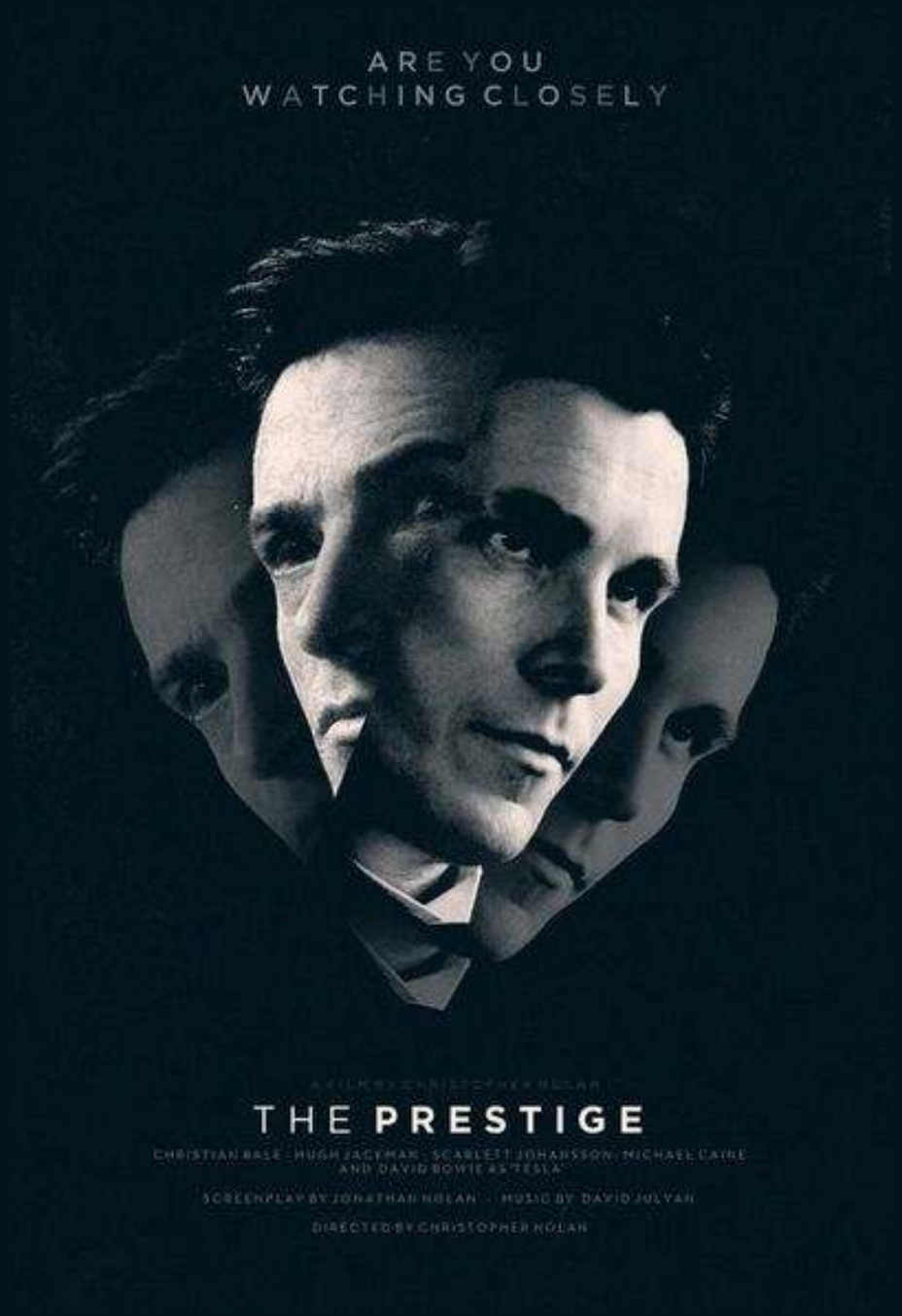
Hot Spot

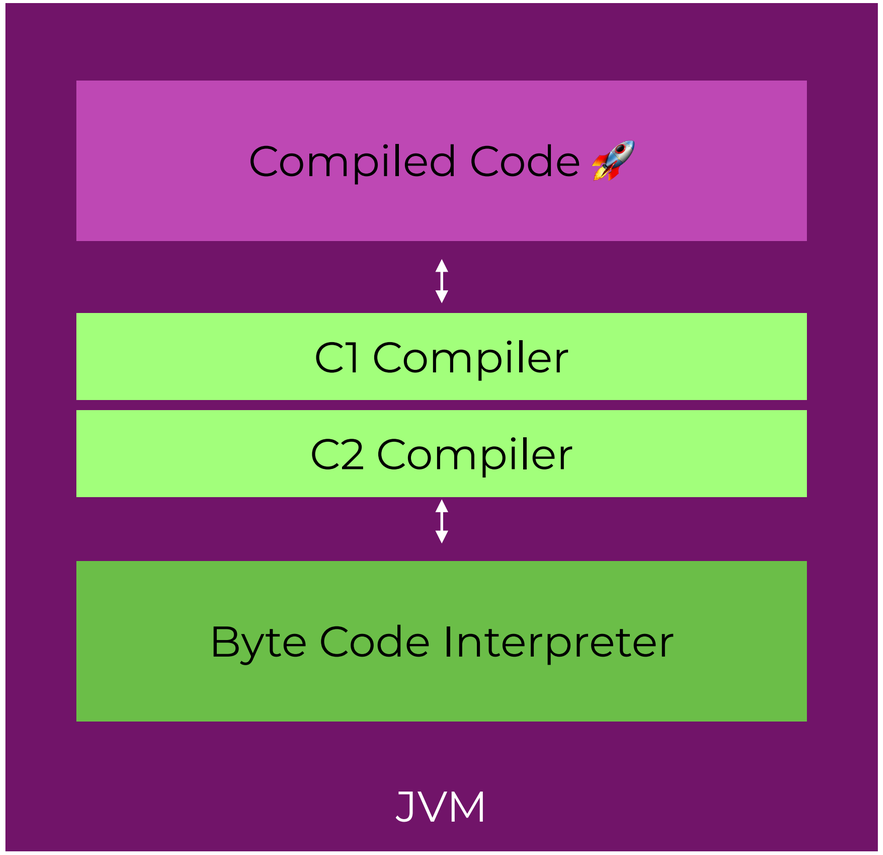
```
0: iload_1
1: iload_2
2: iadd
3: ireturn
```

JIT

HotSpot Analysis

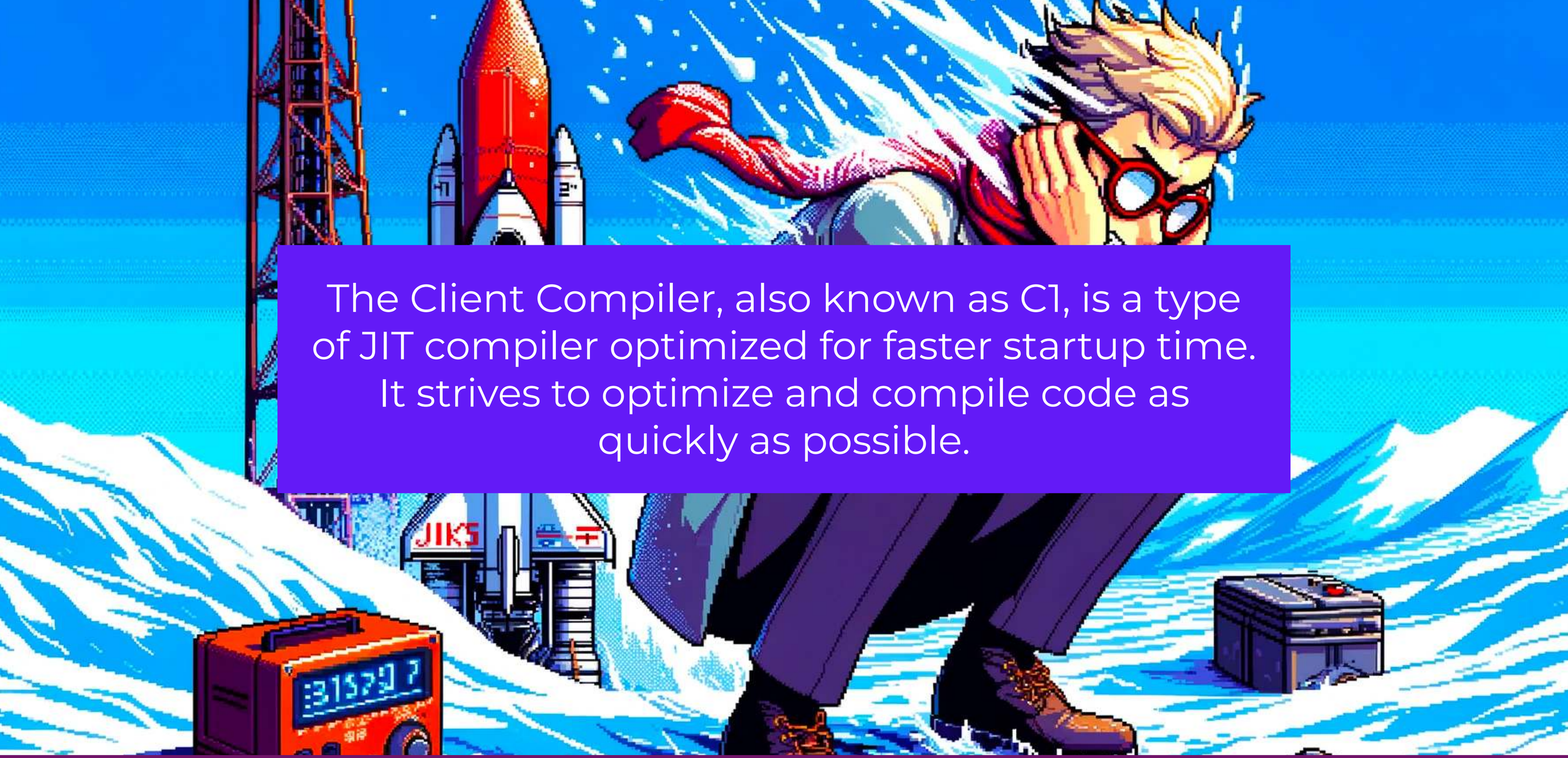
WE NEED TO GO
DEEPER
memegenerator.net


ARE YOU
WATCHING CLOSELY
A FILM BY CHRISTOPHER NOLAN
THE PRESTIGE
CHRISTIAN BALE · HUGH JACKMAN · SCARLETT JOHANSSON · MICHAEL CAINE
AND DAVID BOWIE AS "TESLA"
SCREENPLAY BY JONATHAN NOLAN · MUSIC BY DAVID JULYAN
DIRECTED BY CHRISTOPHER NOLAN


Compiled Code 🚀
C1 Compiler
C2 Compiler
Byte Code Interpreter
JVM

GraalVM

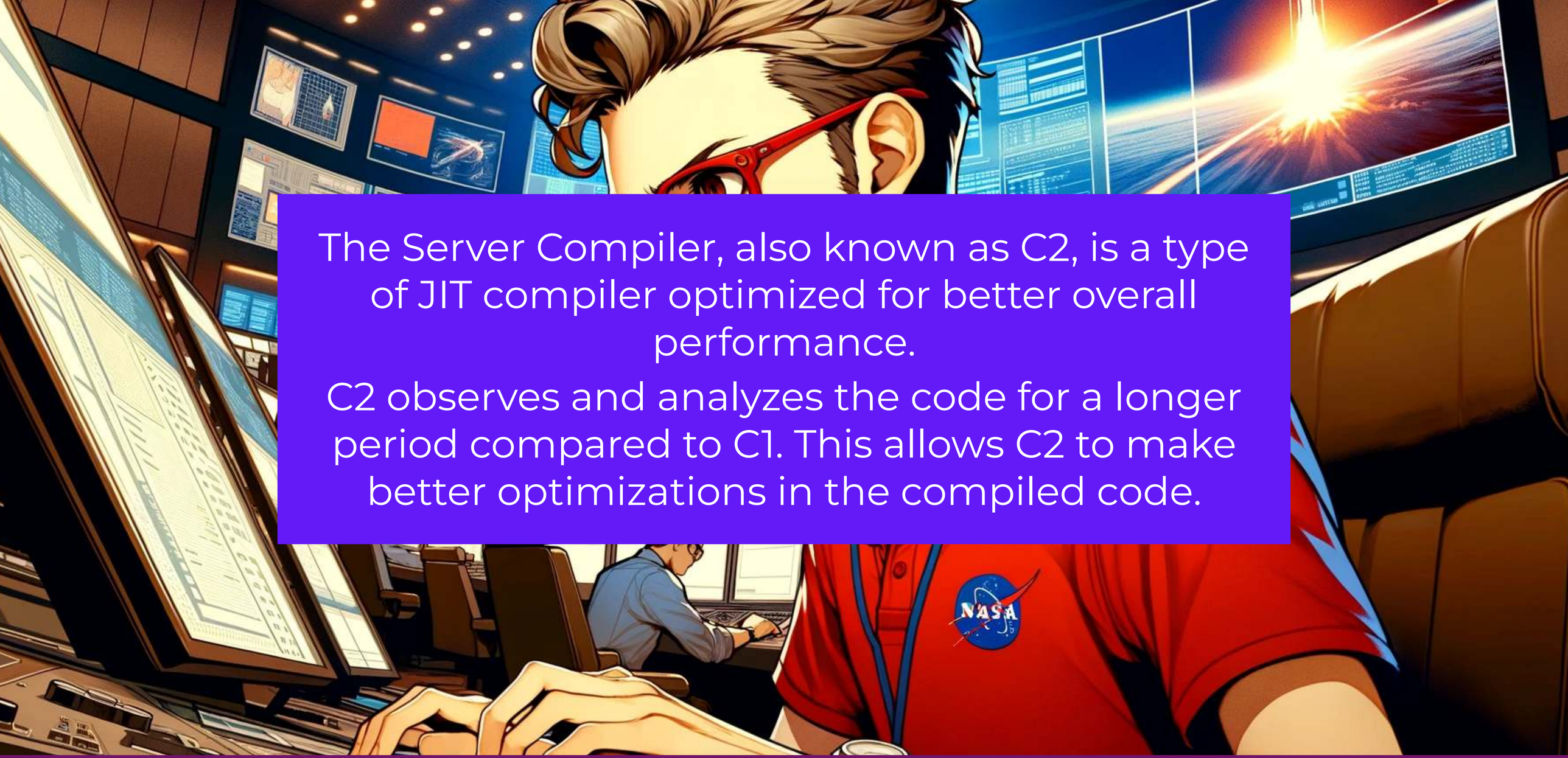
C1 - Client Compiler



The Client Compiler, also known as C1, is a type of JIT compiler optimized for faster startup time. It strives to optimize and compile code as quickly as possible.

C1 - Client Compiler

C2 - Server Compiler

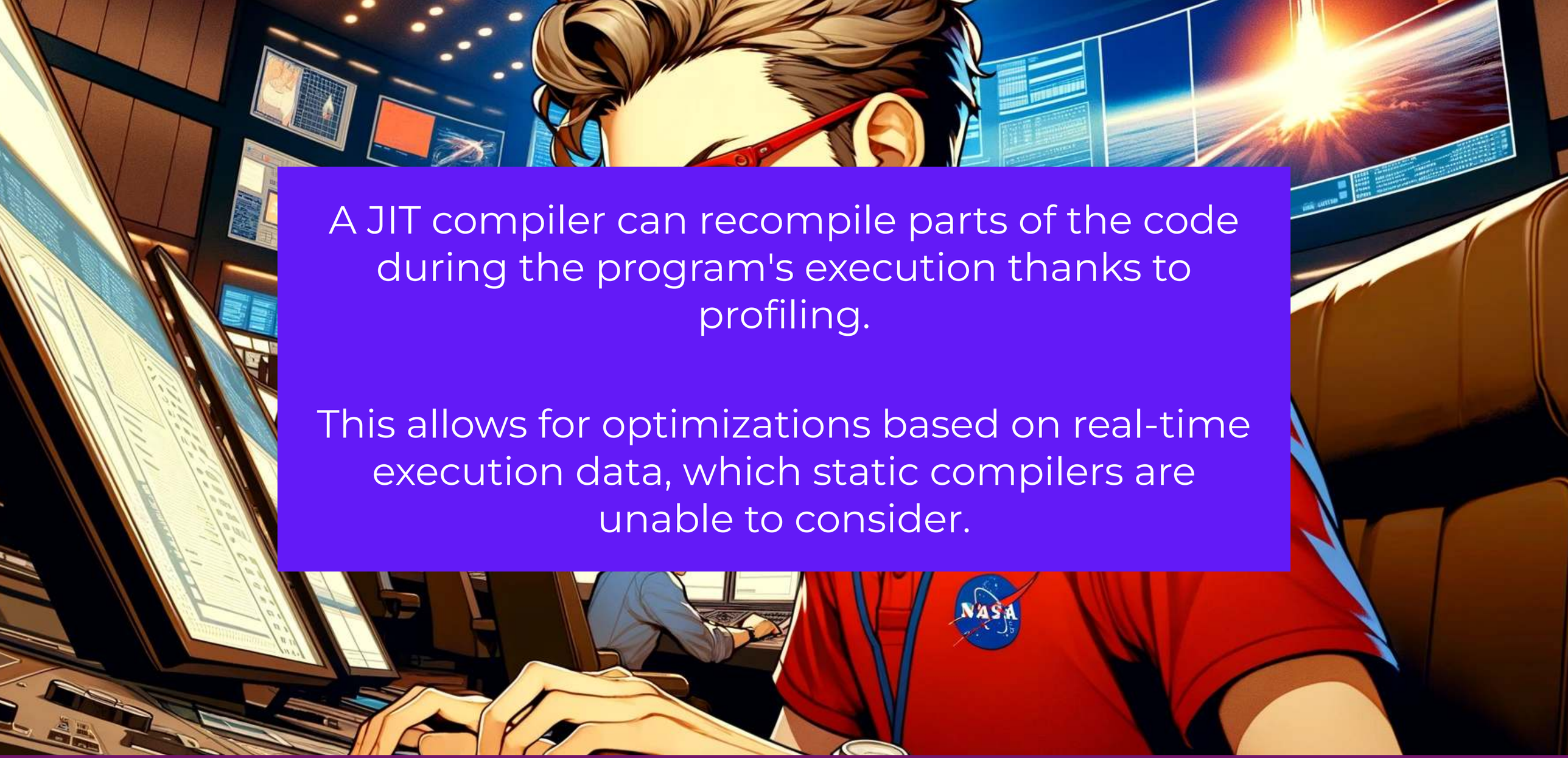


The Server Compiler, also known as C2, is a type of JIT compiler optimized for better overall performance.

C2 observes and analyzes the code for a longer period compared to C1. This allows C2 to make better optimizations in the compiled code.

C2 - Server Compiler

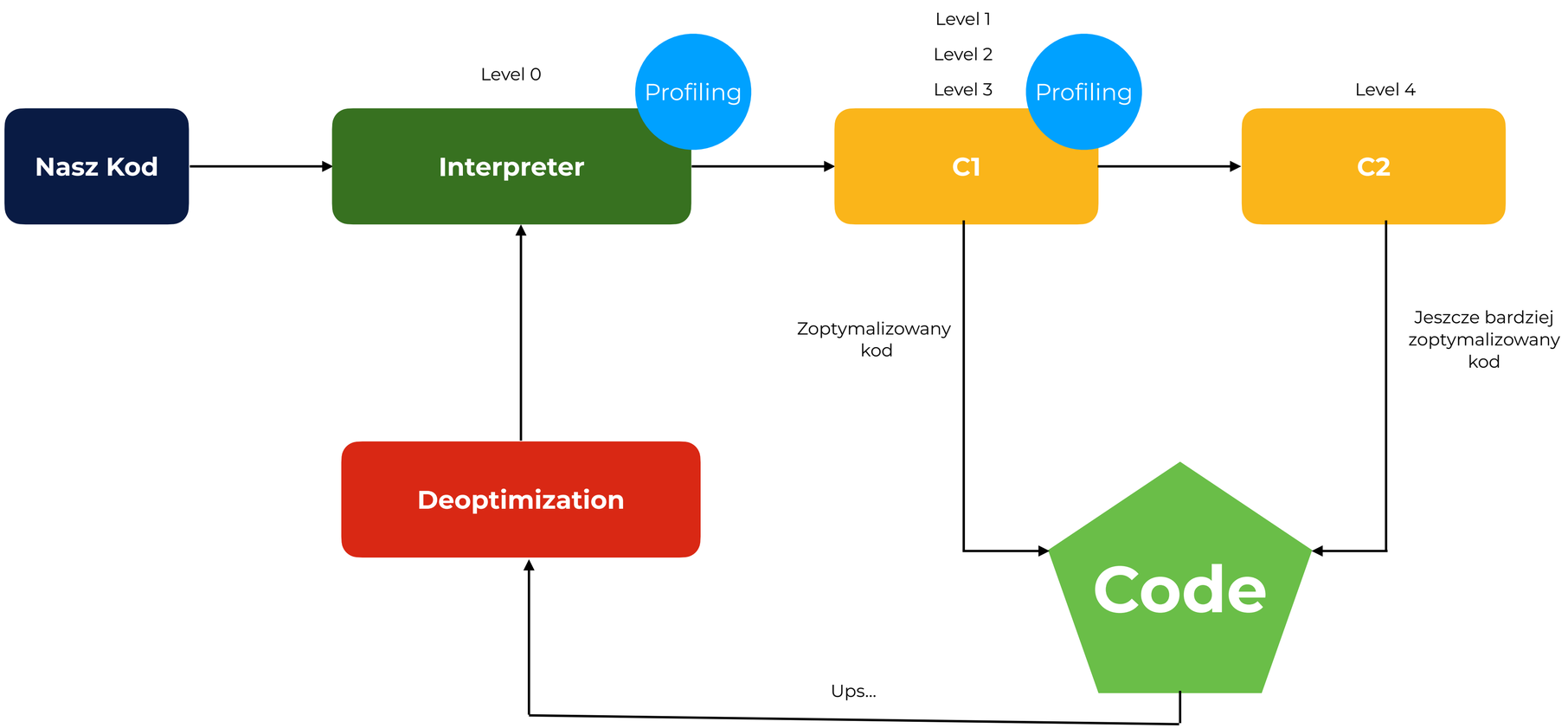
WE NEED TO GO
DEEPER
memegenerator.net



A JIT compiler can recompile parts of the code during the program's execution thanks to profiling.

This allows for optimizations based on real-time execution data, which static compilers are unable to consider.

Dynamic optimization



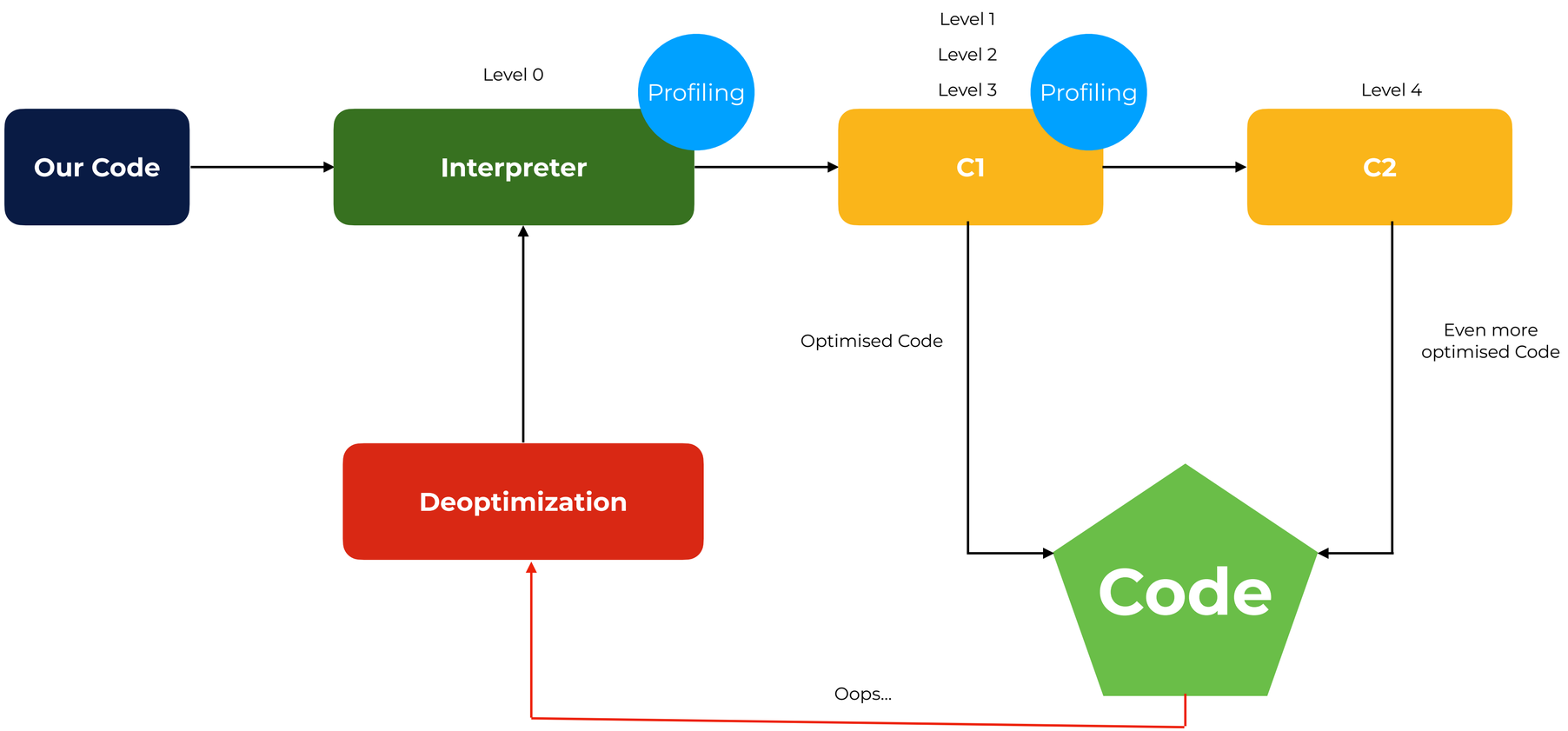
Tiered Compilation

Speculative optimization

Speculative optimization

```java
interface NumberAdder {
    void addRandomNumber(int base);
}

class SimpleAdder implements NumberAdder {
    public void addRandomNumber(int base) {
        double result = Math.random() + base;
    }
}

class LoggingAdder implements NumberAdder {
    public void addRandomNumber(int base) {
        double result = Math.random() + base;
        System.out.println("Computed value: " + result);
    }
}
```

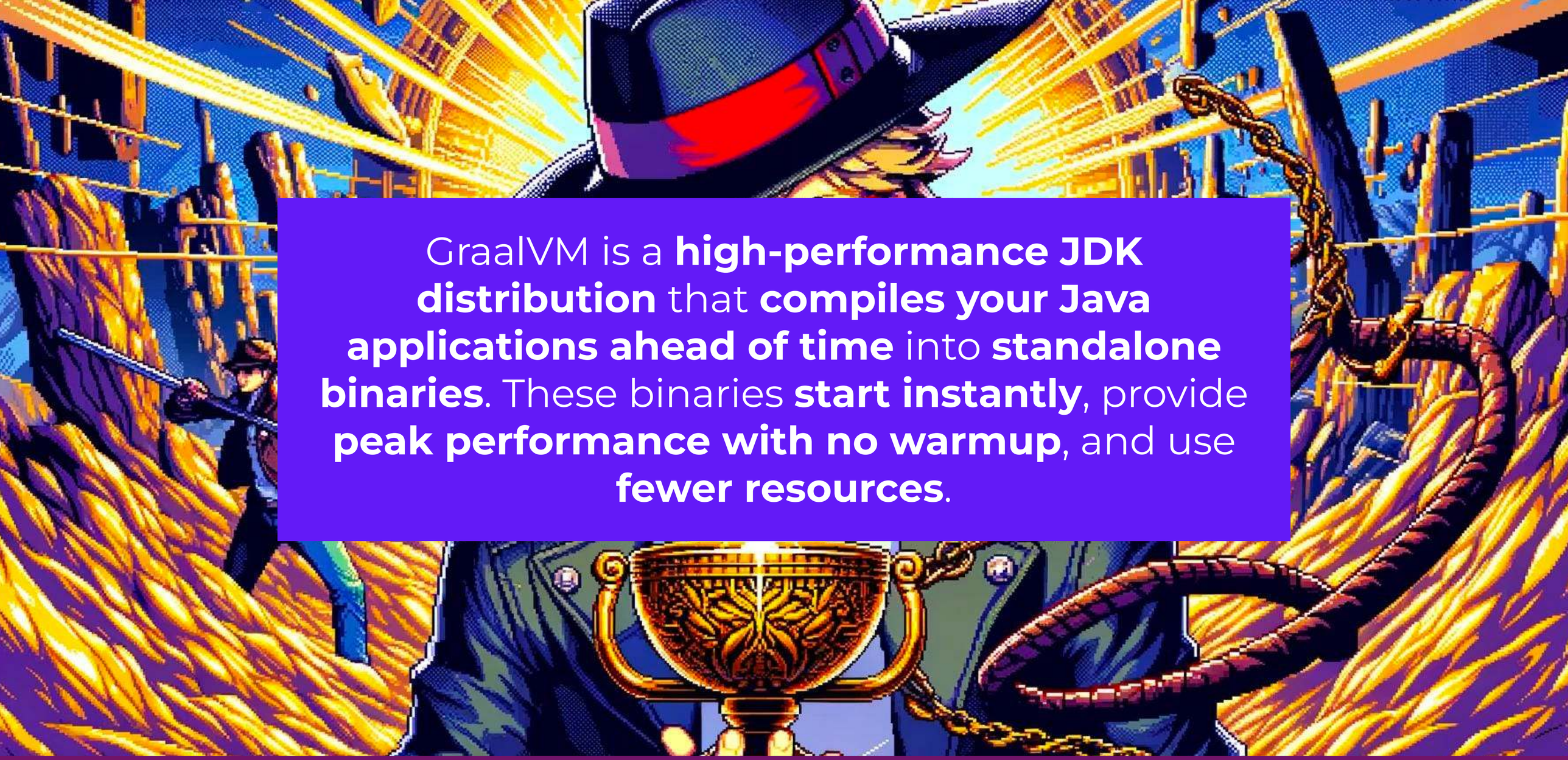
```java
public class DeoptimizationExample {
    public static void main(String[] args) {
        NumberAdder adder = new SimpleAdder();

        for (int i = 0; i < 10000; i++) {
            if (i == 5000) {
                adder = new LoggingAdder();
            }
            adder.addRandomNumber(10);
        }
    }
}

interface NumberAdder {
    void addRandomNumber(int base);
}

class SimpleAdder implements NumberAdder {
    public void addRandomNumber(int base) {
        double result = Math.random() + base;
    }
}

class LoggingAdder implements NumberAdder {
    public void addRandomNumber(int base) {
        double result = Math.random() + base;
        System.out.println("Computed value: " + result);
    }
}
```



Tiered Compilation

Complex, isn't it?

GraalVM



GraalVM is a **high-performance JDK distribution** that **compiles your Java applications ahead of time** into **standalone binaries**. These binaries **start instantly**, provide **peak performance with no warmup**, and use **fewer resources**.

GraalVM

One VM to Rule Them All

Thomas Würthinger* Christian Wimmer* Andreas Wöß† Lukas Stadler†
Gilles Duboscq† Christian Humer† Gregor Richards§ Doug Simon* Mario Wolczko*

*Oracle Labs †Institute for System Software, Johannes Kepler University Linz, Austria §S³ Lab, Purdue University
{thomas.wuerthinger, christian.wimmer, doug.simon, mario.wolczko}@oracle.com
{woess, stadler, duboscq, christian.humer}@ssw.jku.at gr@purdue.edu

Abstract

Building high-performance virtual machines is a complex and expensive undertaking; many popular languages still have low-performance implementations. We describe a new approach to virtual machine (VM) construction that amortizes much of the effort in initial construction by allowing new languages to be implemented with modest additional effort. The approach relies on abstract syntax tree (AST) interpretation where a node can rewrite itself to a more specialized or more general node, together with an optimizing compiler that exploits the structure of the interpreter. The compiler uses speculative assumptions and deoptimization in order to produce efficient machine code. Our initial experience suggests that high performance is attainable while preserving a modular and layered architecture, and that new high-

These implementations can be characterized in the following way:

- Their performance on typical applications is within a small multiple (1-3x) of the best statically compiled code for most equivalent programs written in an unsafe language such as C.
- They are usually written in an unsafe, systems programming language (C or C++).
- Their implementation is highly complex.
- They implement a single language, or provide a bytecode interface that advantages a narrow set of languages to the detriment of other languages.

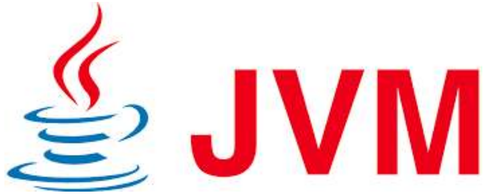
In contrast, there are numerous languages that are popu-

These implementations can be characterized in the following way:

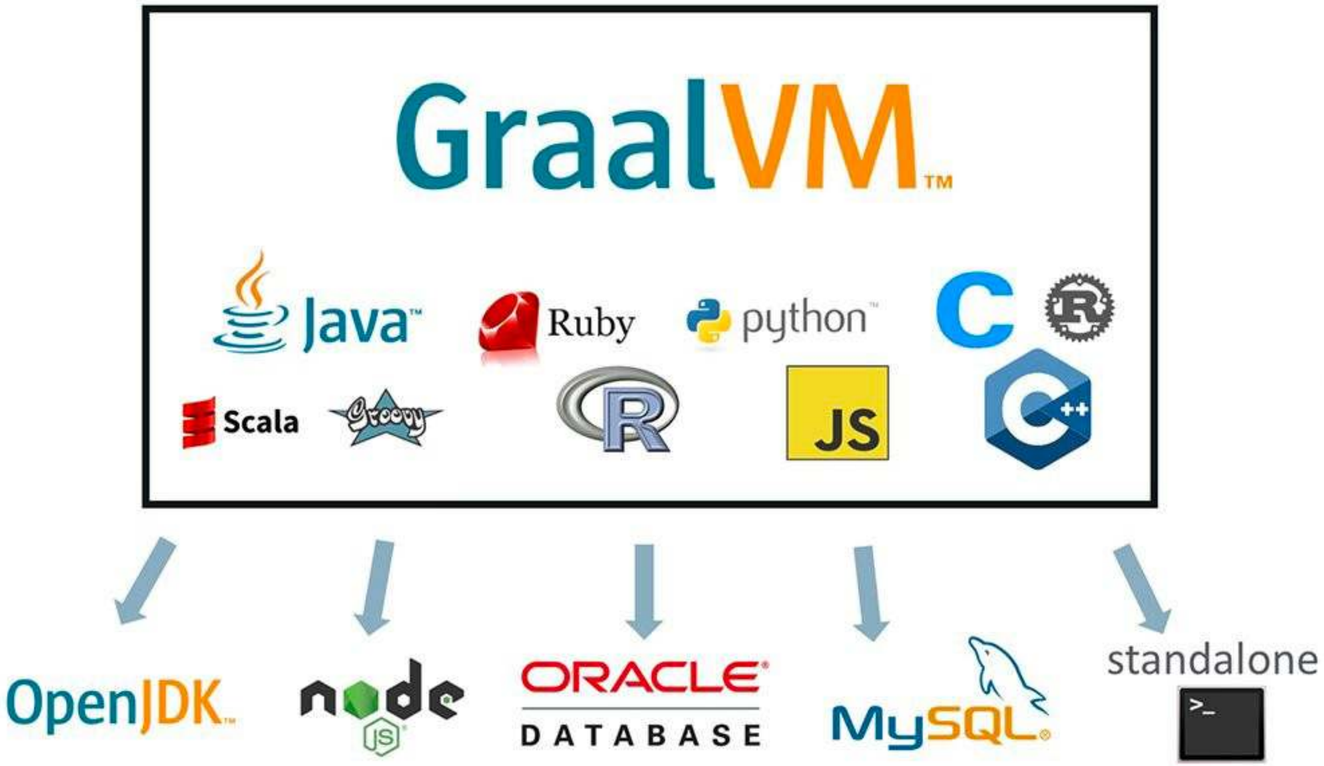
- Their performance on typical applications is within a small multiple (1-3x) of the best statically compiled code for most equivalent programs written in an unsafe language such as C.
- They are usually written in an unsafe, systems programming language (C or C++).
- Their implementation is highly complex.
- They implement a single language, or provide a bytecode interface that advantages a narrow set of languages to the detriment of other languages.

- Their implementation is highly complex.
- They implement a single language, or provide a bytecode interface that advantages a narrow set of languages to the detriment of other languages.

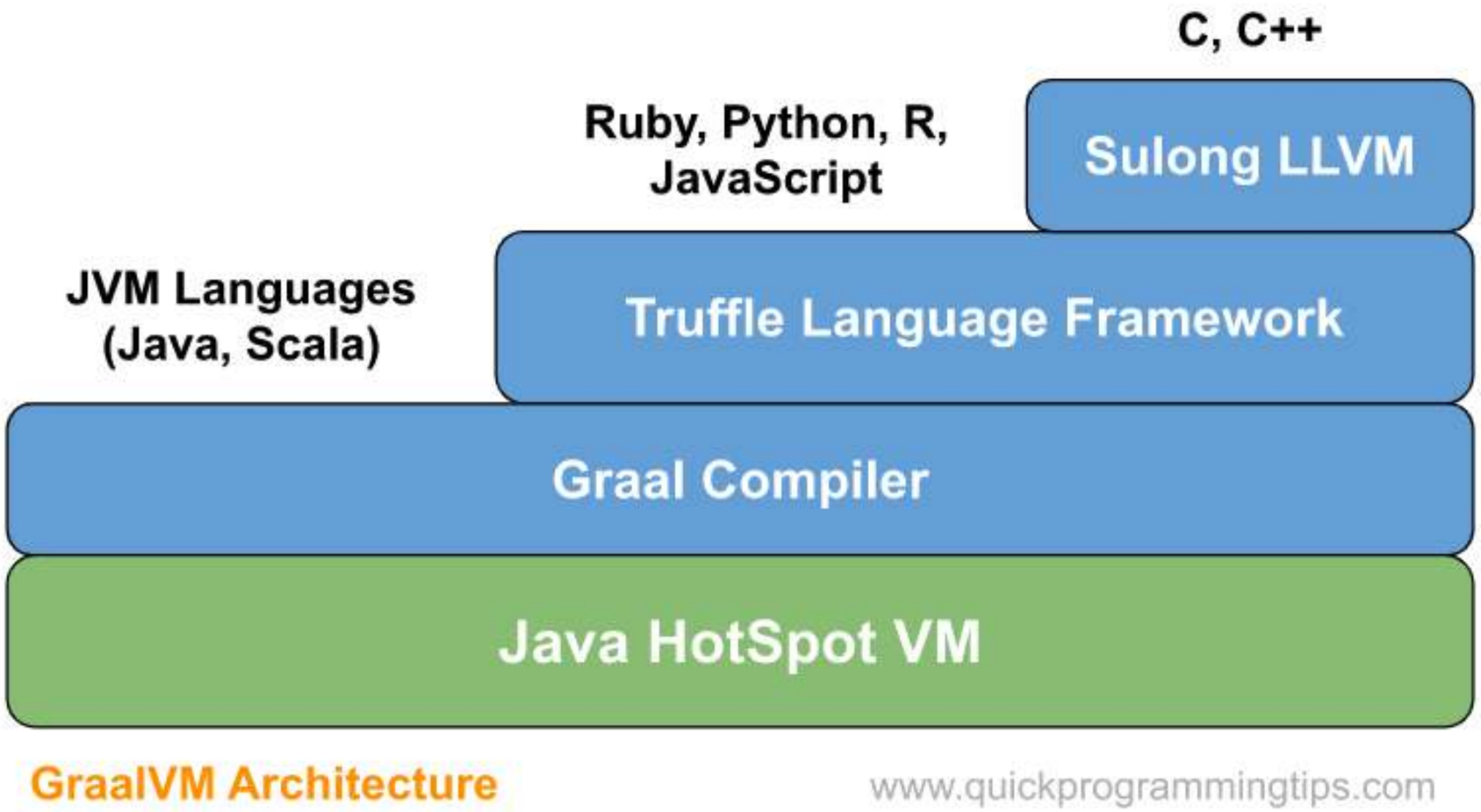
In contrast, there are numerous languages that are popular, have been around for about 20 years, and yet still have no implementations that even approach this level of performance. Examples are PHP, Python, Ruby, R, Perl, MATLAB, Smalltalk, and APL. The *computer language benchmarks game* [20] provides some insights into the performance of various languages.



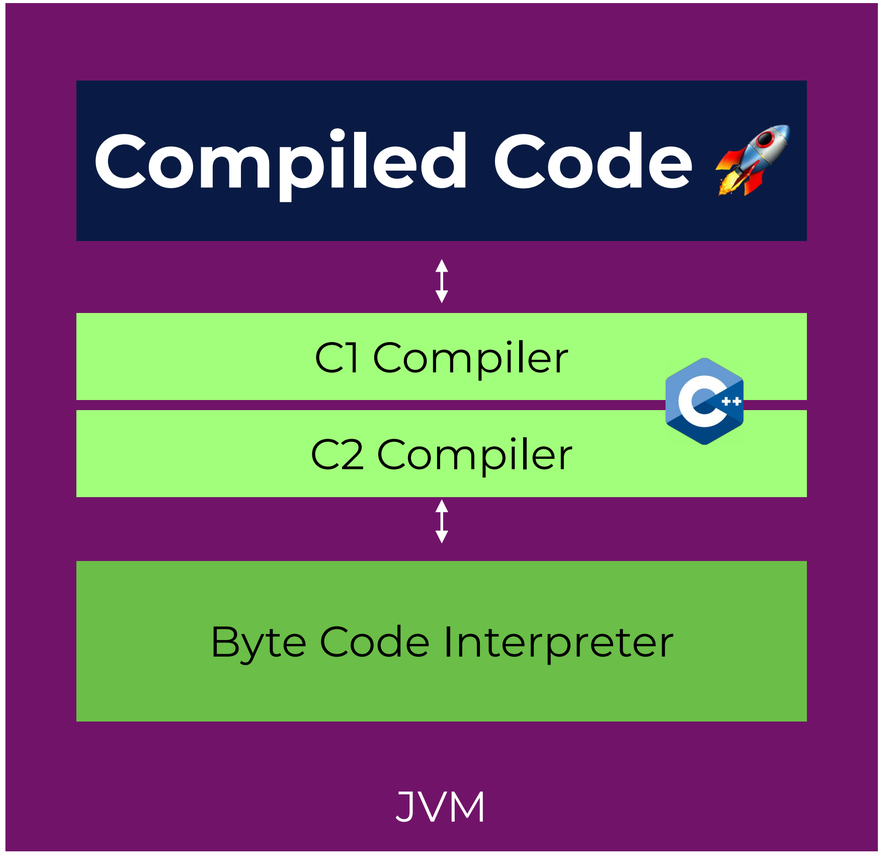
GraalVM


GraalVM
Java
Ruby
python
C
Scala
Groovy
R
JS
C++
OpenJDK
node
ORACLE
DATABASE
MySQL
standalone

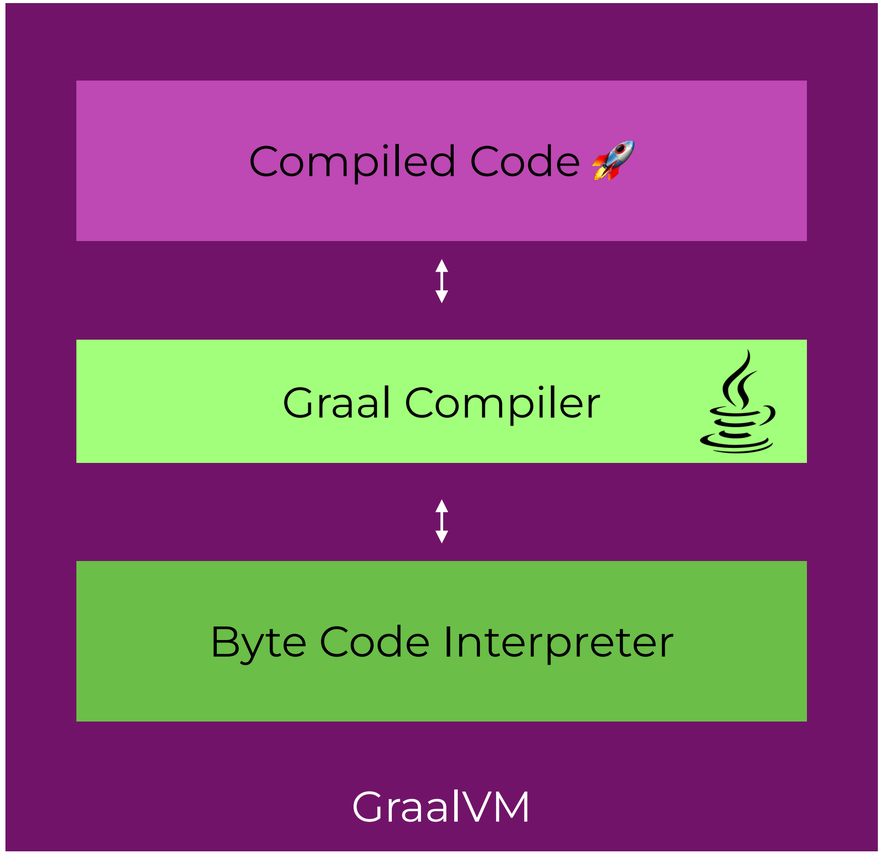
GraalVM


C, C++
Ruby, Python, R, JavaScript
Sulong LLVM
JVM Languages (Java, Scala)
Truffle Language Framework
Graal Compiler
Java HotSpot VM
GraalVM Architecture
www.quickprogrammingtips.com

GraalVM



GraalVM in Just-In-Time (JIT) Compilation Mode



GraalVM in Just-In-Time (JIT) Compilation Mode

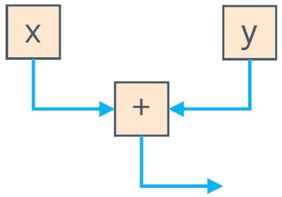
Hot Spot

```
0: iload_1
1: iload_2
2: iadd
3: ireturn
```

JIT

HotSpot Analysis

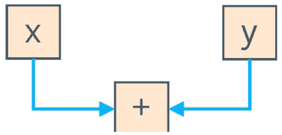
```
0: iload_1
1: iload_2
2: iadd
3: ireturn
```



Graal Compiler

GraalVM in Just-In-Time (JIT) Compilation Mode

```
0: iload_1

1: i

2: i

3: i
```



Second, Graal evaluates the constant objects when it parses the bytecodes and presents those as concrete Java objects to the analysis. Hence, the Graal IR gives a more concrete and easier to use representation of the analyzed code than the bytecode. The bytecode preprocessing step reduces the code complexity and enables the points-to analysis to save some iterations.

GraalVM in Just-In-Time (JIT) Compilation Mode

Which Version of JDK Should I Use?

English 한국어

To build and run Java applications, a Java Compiler, Java Runtime Libraries, and a Virtual Machine are required that implement the Java Platform, Standard Edition ("Java SE") specification.

The OpenJDK is the open source reference implementation of the Java SE Specification, but it is only the source code. Binary distributions are provided by different vendors for a number of supported platforms. These distributions differ in licenses, commercial support, supported platforms, and update frequency.

This site gives independent, yet opinionated recommendations.

GraalVM in Just-In-Time (JIT) Compilation Mode

JEP 243: Java-Level JVM Compiler Interface

Owner	John Rose
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	9
Component	hotspot / compiler
Discussion	hotspot dash compiler dash dev at openjdk dot java dot net
Effort	M
Duration	M
Reviewed by	Douglas Simon, Mikael Vidstedt, Thomas Wuerthinger, Vladimir Kozlov
Endorsed by	Mikael Vidstedt
Created	2014/10/29 20:43
Updated	2019/09/16 19:05
Issue	8062493

Summary

Develop a Java based JVM compiler interface (JVMCI) enabling a compiler written in Java to be used by the JVM as a dynamic compiler.

Goals

- Allow a Java component programmed against the JVMCI to be loaded at runtime and used by the JVM’s compile broker.
- Allow a Java component programmed against the JVMCI to be loaded at runtime and used by trusted Java code to install machine code in the JVM that can be called via a Java reference to the installed code.

JEP 295: Ahead-of-Time Compilation

Owner	Vladimir Kozlov
Type	Feature
Scope	Implementation
Status	Closed / Delivered
Release	9
Component	hotspot / compiler
Discussion	hotspot dash compiler dash dev at openjdk dot java dot net
Effort	M
Duration	M
Reviewed by	John Rose, Mikael Vidstedt
Endorsed by	John Rose
Created	2016/09/15 01:20
Updated	2018/10/05 22:52
Issue	8166089

Summary

Compile Java classes to native code prior to launching the virtual machine.

Goals

- Improve the start-up time of both small and large Java applications, with at most a limited impact on peak performance.
- Change the end user's work flow as little as possible.

GraalVM in Just-In-Time (JIT) Compilation Mode

JEP 317: Experimental Java-Based JIT Compiler

Owner	Igor Veresov
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	10
Component	hotspot / compiler
Discussion	hotspot dash compiler dash dev at openjdk dot java dot net
Effort	M
Duration	M
Reviewed by	John Rose, Mikael Vidstedt, Vladimir Kozlov
Endorsed by	Vladimir Kozlov
Created	2017/10/20 20:03
Updated	2018/03/28 01:58
Issue	8189751

Summary

Enable the Java-based JIT compiler, Graal, to be used as an experimental JIT compiler on the Linux/x64 platform.

Non-Goals

It is not a goal to meet or beat the performance of the existing JIT compilers.

Motivation

Graal, a Java-based JIT compiler, is the basis of the experimental Ahead-of-Time (AOT) compiler introduced in JDK 9. Enabling it to be used as an experimental JIT compiler is one of the initiatives of Project Metropolis, and is the next step in investigating the feasibility of a Java-based JIT for the JDK.

GraalVM in Just-In-Time (JIT) Compilation Mode

JEP 410: Remove the Experimental AOT and JIT Compiler

Owner	Vladimir Kozlov
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	17
Component	hotspot / compiler
Discussion	hotspot dash compiler dash dev at openjdk dot java dot net
Effort	S
Duration	S
Reviewed by	Mikael Vidstedt
Created	2021/03/10 02:59
Updated	2021/08/05 02:44
Issue	8263327

Summary

Remove the experimental Java-based ahead-of-time (AOT) and just-in-time (JIT) compiler. This compiler has seen little use since its introduction and the effort required to maintain it is significant. Retain the experimental Java-level JVM compiler interface (JVMCI) so that developers can continue to use externally-built versions of the compiler for JIT compilation.

Motivation

Ahead-of-time compilation (the `jaotc` tool) was incorporated into JDK 9 as an experimental feature via JEP 295. The `jaotc` tool uses the Graal compiler, which is itself written in Java, for AOT compilation.

The Graal compiler was made available as an experimental JIT compiler in JDK 10 via JEP 317.

We have seen little use of these experimental features since they were introduced, and the effort required to maintain and enhance them is significant. These features were not included in the JDK 16 builds published by Oracle, and no one complained.

GraalVM in Just-In-Time (JIT) Compilation Mode

Galahad

Galahad's primary goal is to contribute Java-related GraalVM technologies to the OpenJDK Community and prepare them for possible incubation in a JDK main-line release.

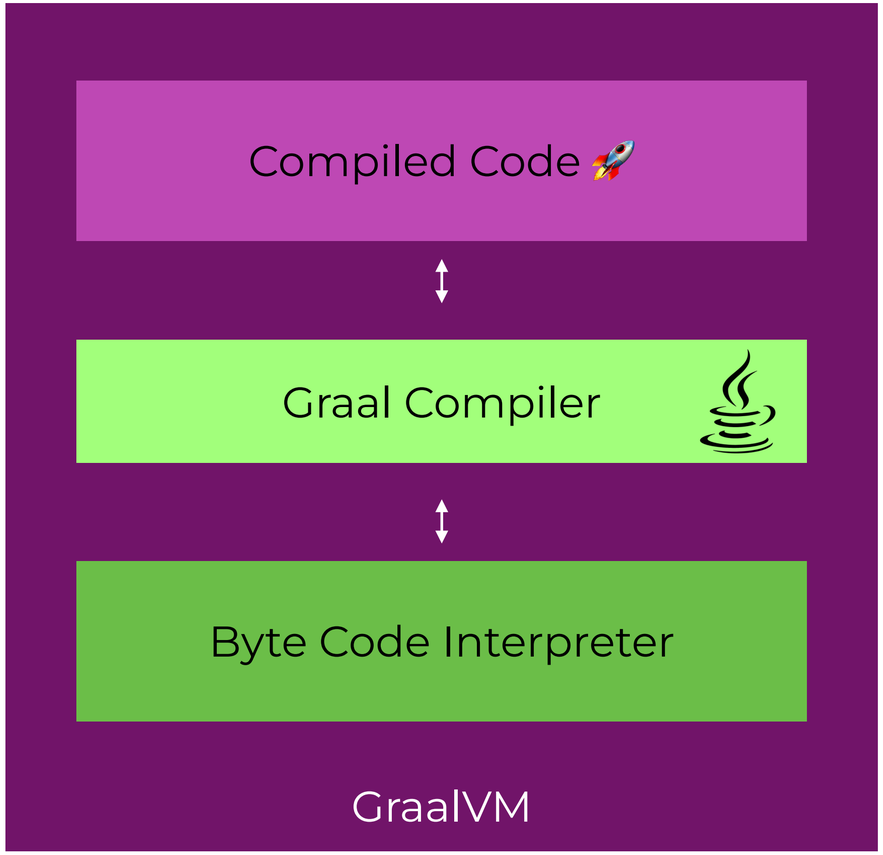
This Project is sponsored by the HotSpot Group.

Community

- Members
- Mailing list
  - galahad-dev (archives)
- Links
  - Repository

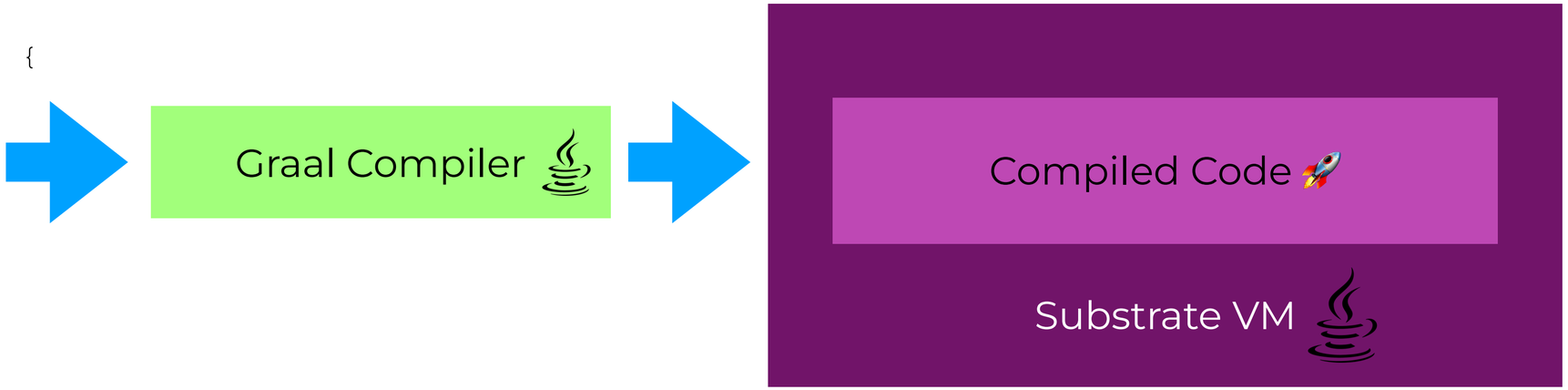
GraalVM in Just-In-Time (JIT) Compilation Mode

GraalVM in Just-In-Time (JIT) Compilation Mode



GraalVM in Ahead-of-Time (AoT) Compilation Mode

```
public static int add(int a, int b) {

    return a + b;

}
```



GraalVM in Ahead-of-Time (AoT) Compilation Mode

The Substrate VM is low-footprint, fast start-up, embeddable virtual machine (VM) for and written in a subset of Java. It serves as an implementation basis for our hybrid memory management technique.

Closed World Assumption

Closed World Assumption

Going AOT: Everything you need to know about GraalVM for Java

Session type: ARCHITECTURE

In the past few years GraalVM got widely adopted by the Java ecosystem, and now it's hard to imagine starting a new project, especially for the cloud, without at least considering going native. However, taking a new technology to production requires research and preparation. What is the best way to build and deploy such native executables? Once deployed, how can I monitor them? Can I test them as I do with regular Java applications? What if startup is less important, how do I optimize for peak performance and latency? How do I use frameworks and libraries? This session will cover all of those steps and illustrate them with examples. We'll also have plenty of time for questions – bring them all!

Alina Yurenko

Alina is a developer advocate for GraalVM at Oracle Labs, a research & development organization at Oracle. Loves both programming & natural languages, compilers, and open source.

Finally, we can move on to Truffle

In contrast, there are numerous languages that are popular, have been around for about 20 years, and yet still have no implementations that even approach this level of performance. Examples are PHP, Python, Ruby, R, Perl, MATLAB, Smalltalk, and APL. The *computer language benchmarks game* [20] provides some insights into the performance of various languages.

Self-Optimizing AST Interpreters

Thomas Würthinger* Andreas Wöß† Lukas Stadler† Gilles Duboscq†
Doug Simon* Christian Wimmer*

*Oracle Labs †Institute for System Software, Johannes Kepler University Linz, Austria
thomas.wuerthinger@oracle.com woess@ssw.jku.at stadler@ssw.jku.at duboscq@ssw.jku.at
doug.simon@oracle.com christian.wimmer@oracle.com

Abstract

An abstract syntax tree (AST) interpreter is a simple and natural way to implement a programming language. However, it is also considered the slowest approach because of the high overhead of virtual method dispatch. Language implementers therefore define bytecodes to speed up interpretation, at the cost of introducing inflexible and hard to maintain bytecode formats. We present a novel approach to implementing AST interpreters in which the AST is modified during interpretation to incorporate type feedback. This tree rewriting is a general and powerful mechanism to optimize many constructs common in dynamic programming languages. Our system is implemented in Java™ and uses the static typing and primitive data types of Java elegantly to avoid the cost of boxed representations of primitive values in dynamic programming languages.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments, Optimization

General Terms Algorithms, Languages, Performance

Keywords Java, JavaScript, dynamic languages, virtual machine, language implementation, optimization

studied [9, 22]. Bytecodes are a virtual instruction set, i.e, they can be defined freely by language implementers without worrying about hardware constraints. There are two different strategies when defining bytecodes. We believe that both are cumbersome when implementing a new language:

- Language-independent bytecodes aim to provide an instruction set that is suitable for many languages. Java™ bytecodes [17], originally designed primarily for the Java programming language, are now used for many languages. The Common Intermediate Language (CIL) [8], which is used by the .NET system, was specifically designed for multiple languages. Language-independent bytecodes provide only general instructions whose semantics do not match the language semantics exactly. Long bytecode sequences are potentially required to achieve the desired language behavior. For example, Java bytecodes are statically typed, so compiling dynamically typed JavaScript requires frequent type checks. A simple addition in JavaScript requires checking whether the operands are numbers, strings, or arbitrary objects, which all require different handling. Therefore, implementations of JavaScript that run on a Java VM often call a method for a seemingly simple addition.

GraalVM in Truffle Mode

TNF

Faculty of Engineering
and Natural Sciences

TruffleClojure: A self-optimizing AST-Interpreter for Clojure

Master's Thesis

submitted in partial fulfillment of the requirements
for the academic degree

Diplom-Ingenieur

in the Master's Program

Computer Science

GraalVM in Truffle Mode

TNF

Faculty of Engineering
and Natural Sciences

TruffleClojure: A self-optimizing AST-Interpreter for Clojure

Master's Thesis

submitted in partial fulfillment of the requirements
for the academic degree

Diplom-Ingenieur

in the Master's Program

Computer Science

Dynamic type system:

Clojure features a dynamic type system. The Truffle framework was designed for high performance implementations for languages with a dynamic type system. As a consequence, the Clojure programming language could also benefit from the optimizations performed by Truffle-based interpreters.

GraalVM in Truffle Mode

And once again, a side-track

Dynamic vs Static Typing

```
APLUSB   CSECT
         USING  APLUSB,R12
         LR     R12,R15
         OPEN   (MYDATA,INPUT)
LOOP     GET    MYDATA,PG          read a single record
         XDECI  R4,PG              input A
         XDECI  R5,PG+12           input B
         AR     R4,R5              A+B
         XDECO  R4,PG+24           edit A+B
         XPRNT  PG,36              print A+B
         B      LOOP               repeat
ATEND    CLOSE  MYDATA
RETURN   XR     R15,R15
         BR     R14
         LTORG
MYDATA   DCB    LRECL=24,RECFM=FT,EODAD=ATEND,DDNAME=MYFILE
PG       DS     CL24               record
         DC     CL12' '
         YREGS
         END    APLUSB
```

How can we prove that this program works?

CLASSES OF RECURSIVELY ENUMERABLE SETS AND THEIR DECISION PROBLEMS(1)

BY

H. G. RICE

1. **Introduction.** In this paper we consider classes whose elements are recursively enumerable sets of non-negative integers. No discussion of recursively enumerable sets can avoid the use of such classes, so that it seems desirable to know some of their properties. We give our attention here to the properties of complete recursive enumerability and complete recursiveness (which may be intuitively interpreted as decidability). Perhaps our most interesting result (and the one which gives this paper its name) is the fact

RICE'S THEOREM (1957)

EVERY NON-TRIVIAL PROPERTY OF RECURSIVELY ENUMERABLE LANGUAGES IS UNDECIDABLE.

I. FUNDAMENTAL DEFINITIONS

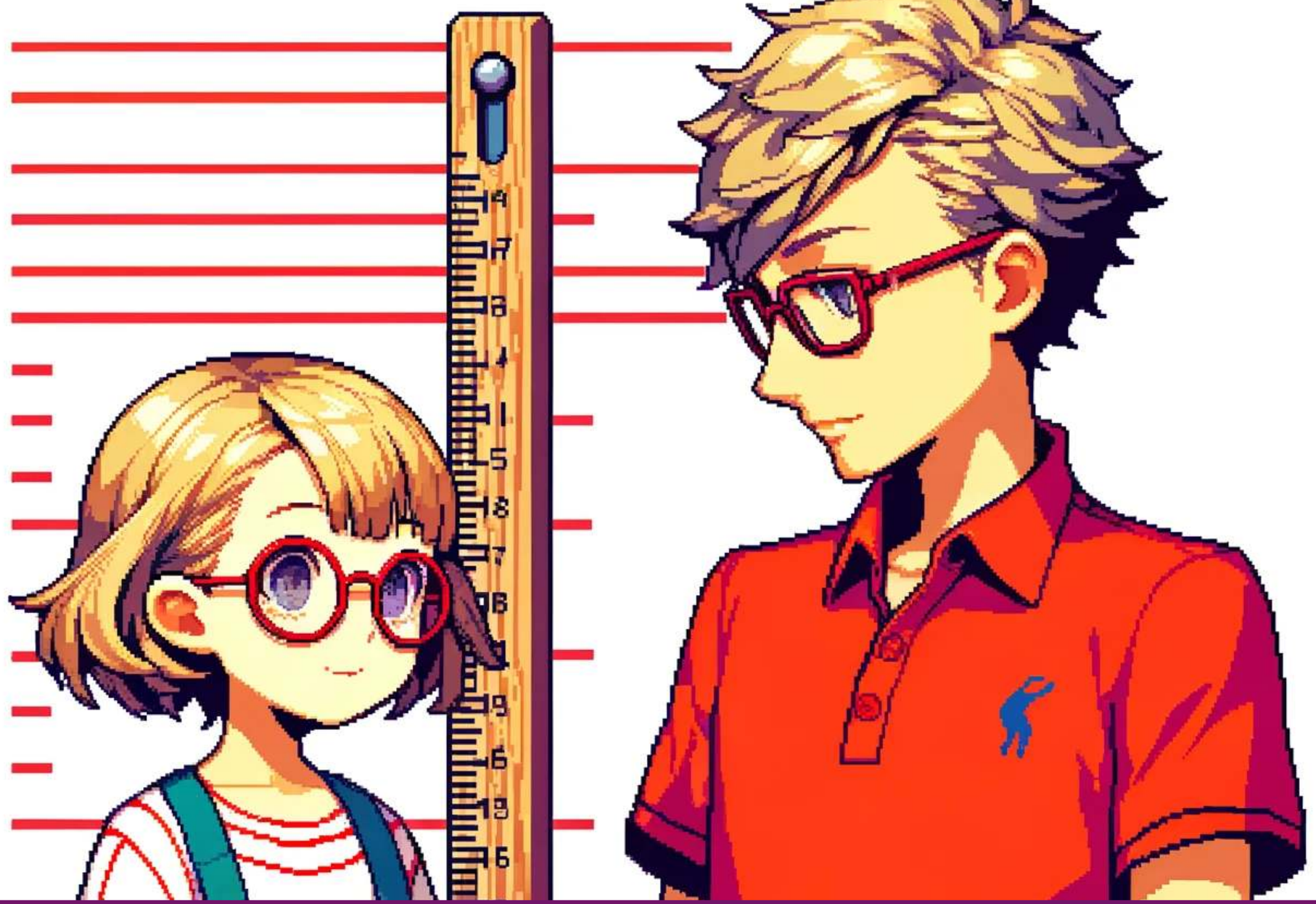
Static typing - A set of rules to follow

```fortran
program fizzbuzz_if
   integer :: i

   do i = 1, 100
      if      (mod(i,15) == 0) then; print *, 'FizzBuzz'
      else if (mod(i,3) == 0) then; print *, 'Fizz'
      else if (mod(i,5) == 0) then; print *, 'Buzz'
      else;                         print *, i
      end if
   end do
end program fizzbuzz_if
```

Fortran (1957)

```fortran
program fizzbuzz_if
   integer :: i

   do i = 1, 100
      if      (mod(i,15) == 0) then; print *, 'FizzBuzz'
      else if (mod(i,3) == 0) then; print *, 'Fizz'
      else if (mod(i,5) == 0) then; print *, 'Buzz'
      else;                         print *, i
      end if
   end do
end program fizzbuzz_if
```

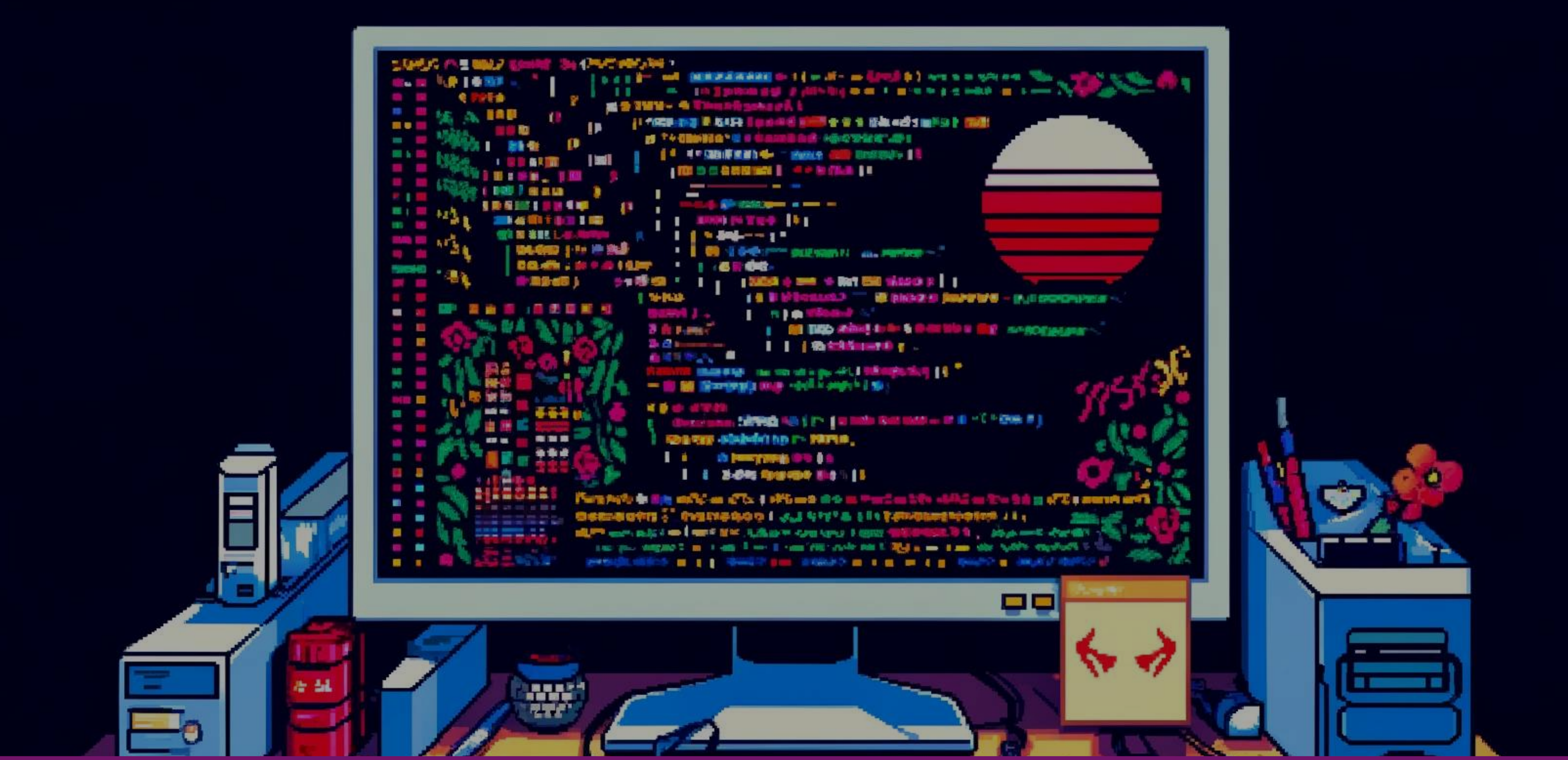
Fortran (1957)



Lower Bound of Correctness

```java
List<Integer> integers = new ArrayList<>();
List rawList = integers;

rawList.add("hello");

try {
    Integer num = integers.get(0);
} catch (ClassCastException e) {
    System.out.println("Error: " + e.getMessage());
}
```

Tradeoff: Easy to use / correct

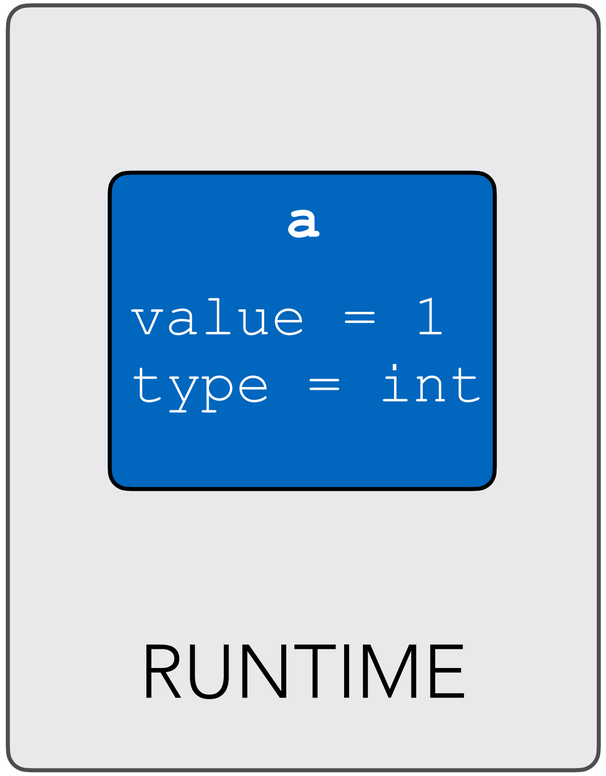


Dynamic typing - Managing types at runtime

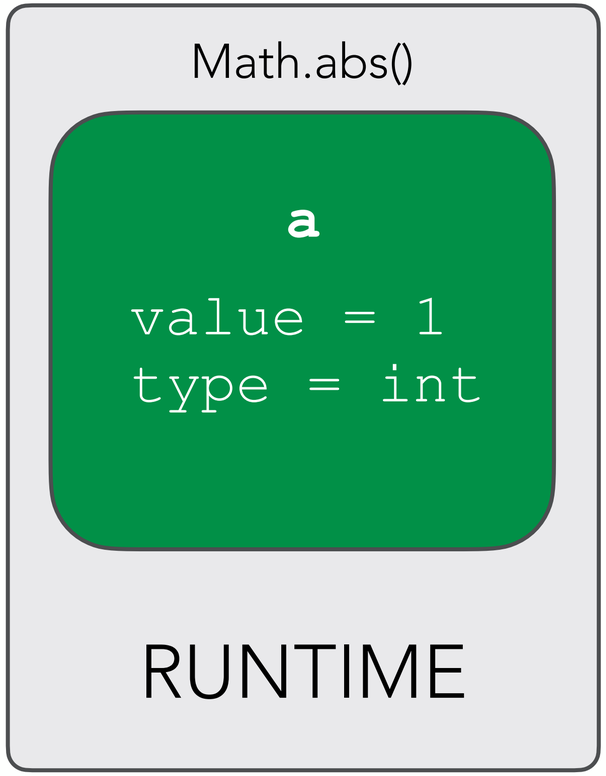
```
function add (a, b) {
return a + b;  }
```

RUNTIME

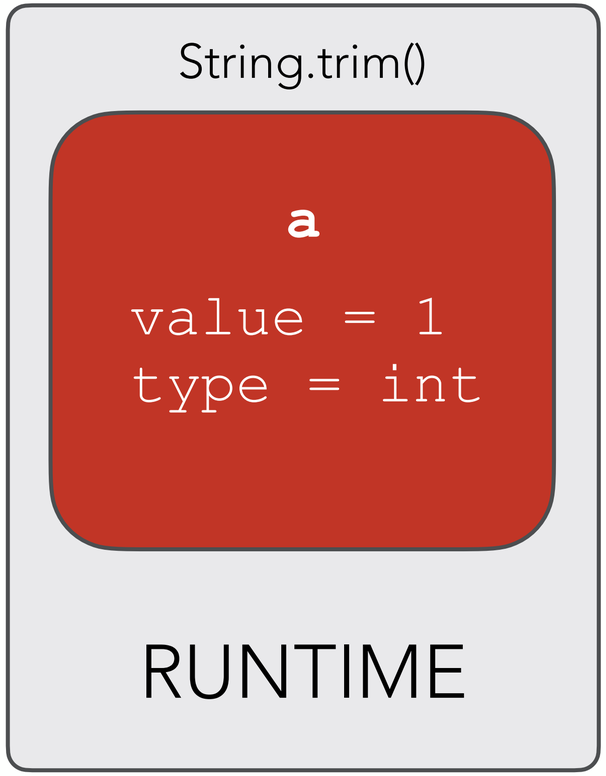
Value in Code



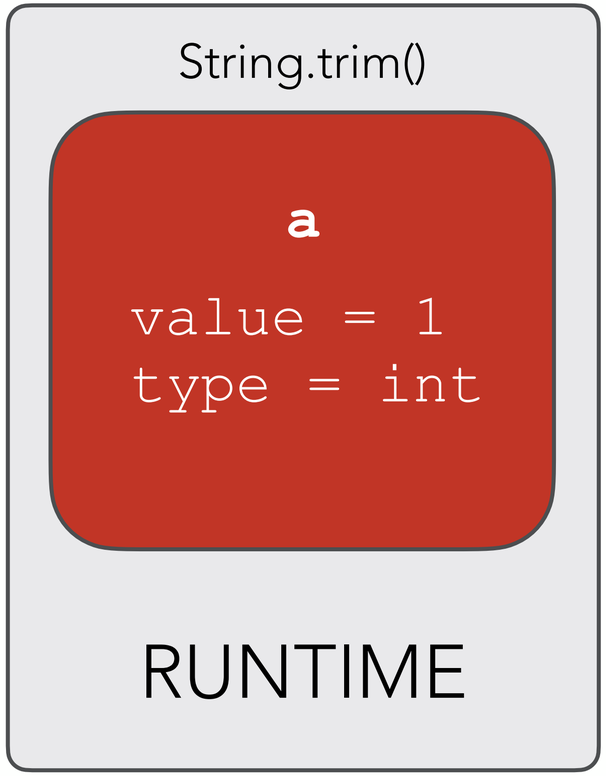
Value in Code



Everything is in Check

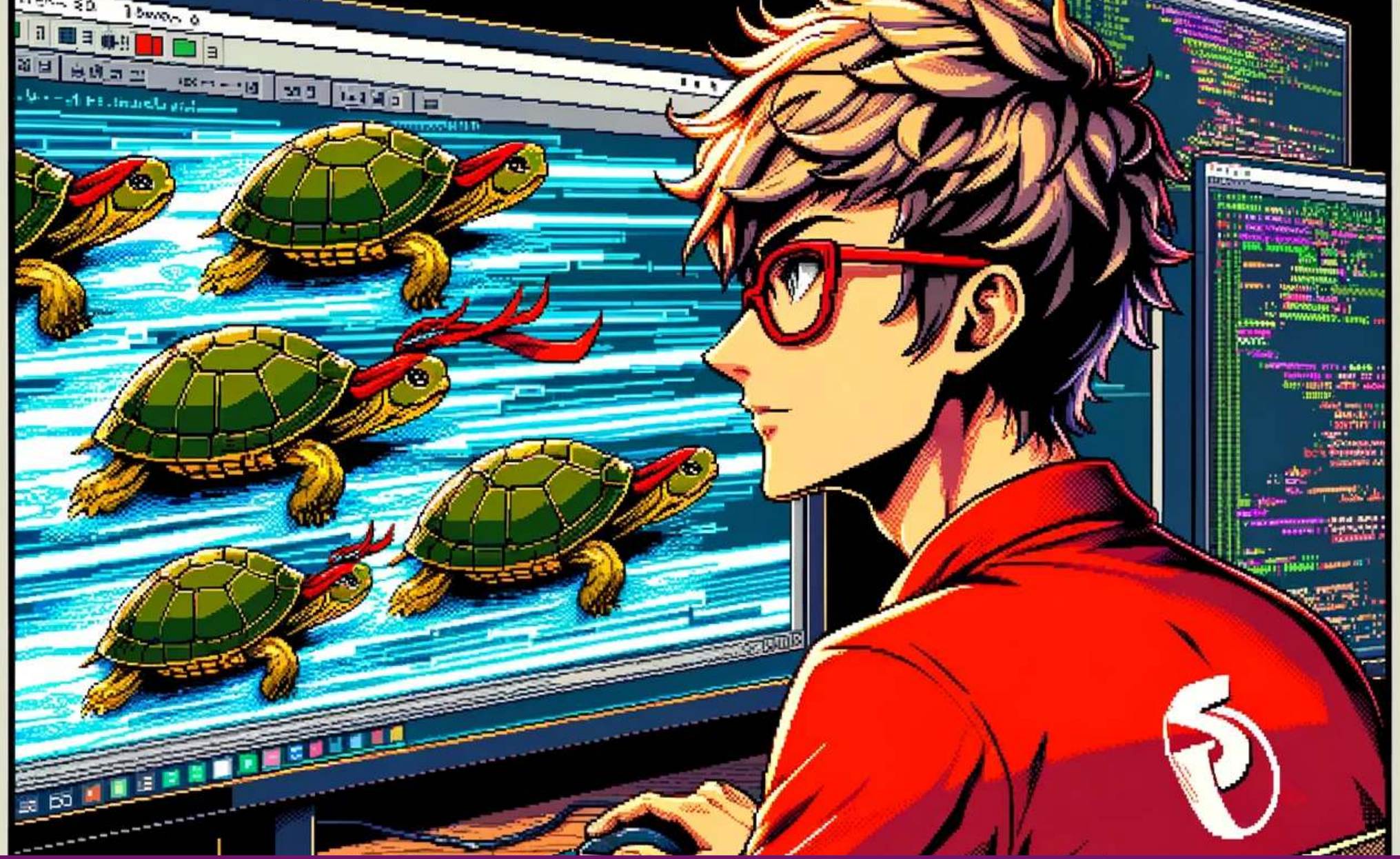


Runtime Exception



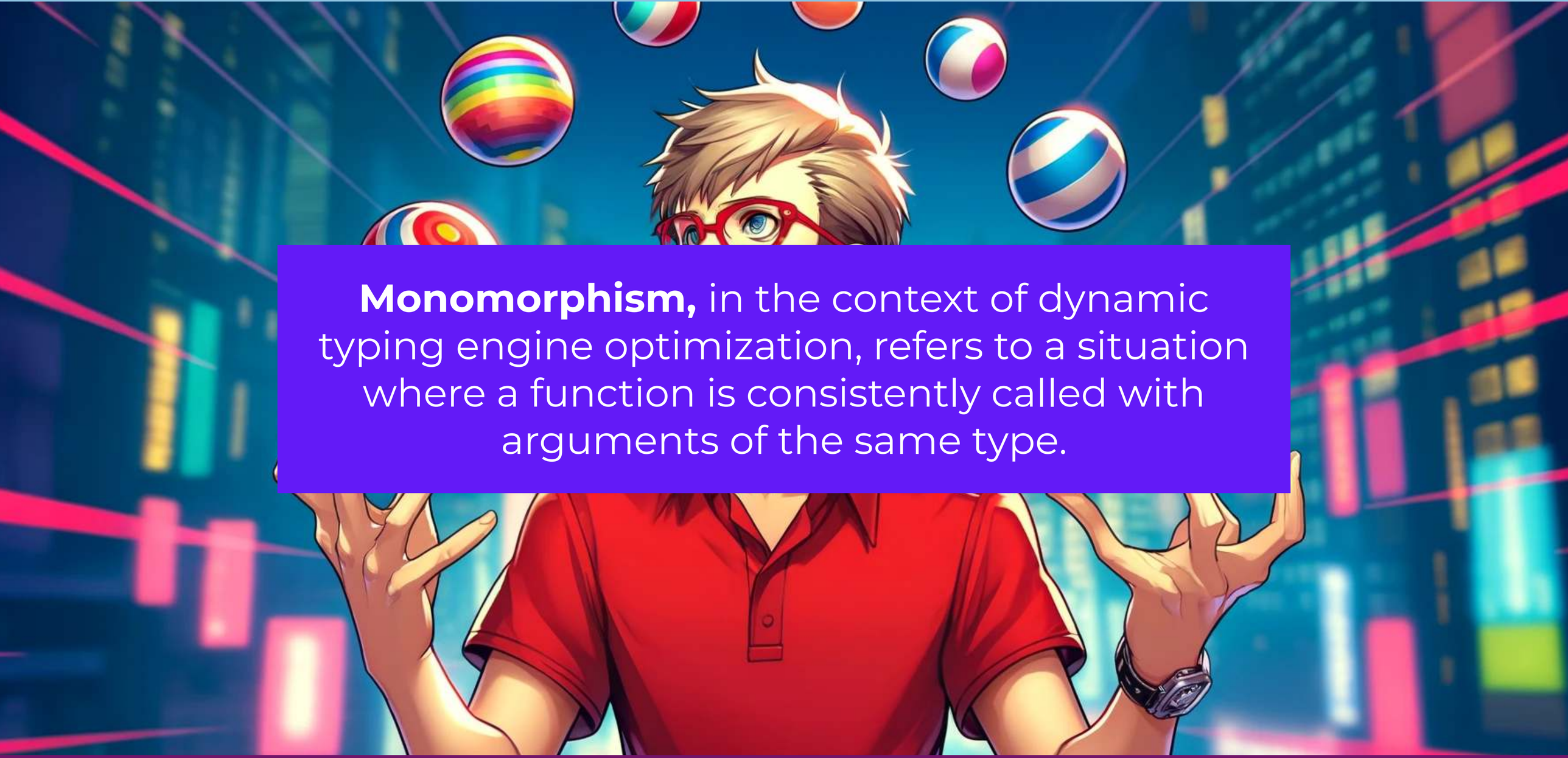
Runtime Exception

I'VE FINALLY FOUND the buggy line
Robotatertotcomics
console.log(_dasObject)
[Object object]
NYEHHH



Performance Overhead

```javascript
function add(x, y) {
    return x + y;
}

// Monomorphic usage
console.log(add(1, 2));
console.log(add(3, 4));

// Polymorphic usage
console.log(add(1, 2));
console.log(add("3", "4"));
```



Monomorphism, in the context of dynamic typing engine optimization, refers to a situation where a function is consistently called with arguments of the same type.

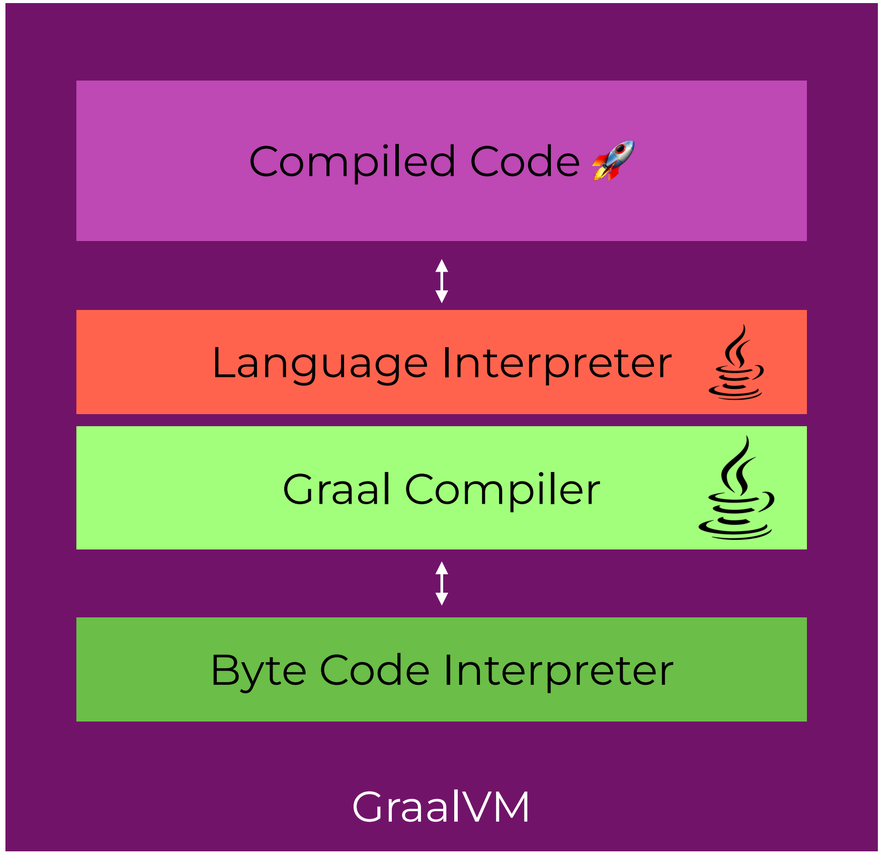
Dynamic Typing

Polymorphism: A function is called with arguments of different types. This requires the engine to handle multiple cases and can lead to less efficient code.

Megamorphism: A function is called with many different types of arguments, making it nearly impossible for the engine to optimize.

Dynamic Typing

Finally... back to Truffle



GraalVM in Truffle Mode

Language Interpreter

GraalVM in Truffle Mode

```java
import com.oracle.truffle.api.dsl.Specialization;
import com.oracle.truffle.api.dsl.TypeSystemReference;
import com.oracle.truffle.api.nodes.Node;
import com.oracle.truffle.api.nodes.NodeInfo;

@TypeSystemReference(MyTypeSystem.class) // Define the types recognized in your language
@NodeInfo(shortName = "add")
public abstract class AddNode extends Node {

    abstract Object execute(Object a, Object b);

    @Specialization
    protected int addIntegers(int a, int b) {
        return a + b;
    }

    @Specialization
    protected double addDoubles(double a, double b) {
        return a + b;
    }

    // Can add more specializations (e.g., for adding strings, booleans, etc.)
}
```

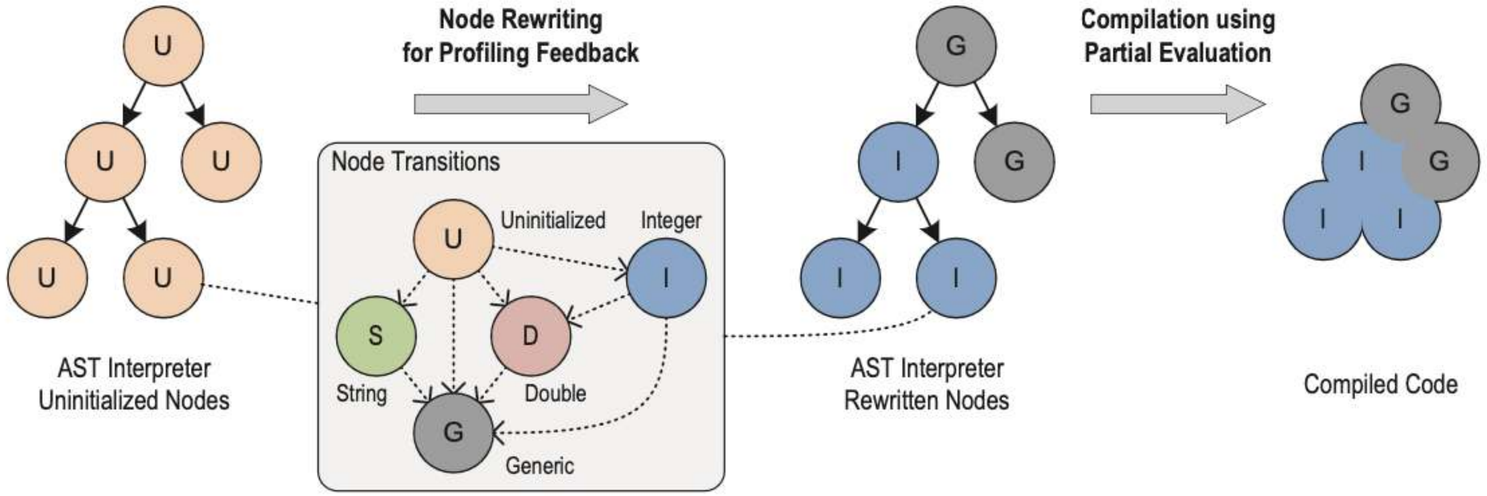


Figure 2. Node rewriting for profiling feedback and partial evaluation lead to aggressively specialized machine code.

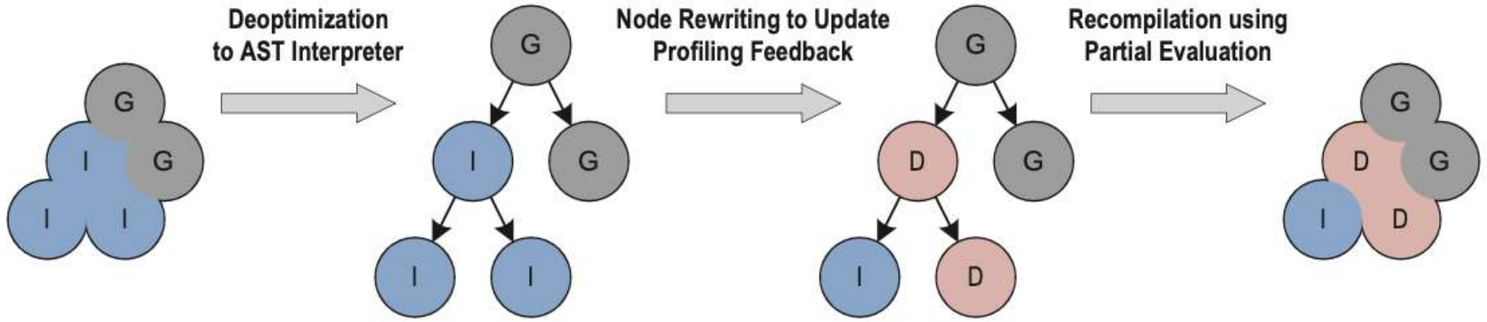


Figure 3. Deoptimization back to the AST interpreter handles speculation failures.

GraalVM

Abstract syntax tree

21 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

For the trees used in linguistics, see parse tree.

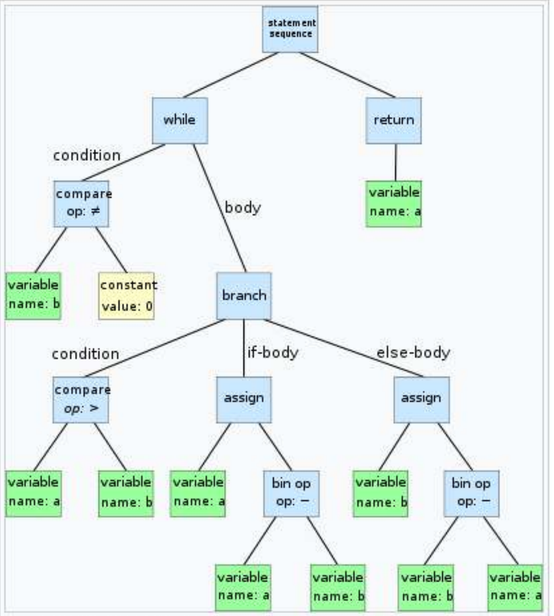
This article includes a list of references, related reading, or external links, **but its sources remain unclear because it lacks inline citations**. Please help improve this article by introducing more precise citations. *(February 2013) (Learn how and when to remove this template message)*

An **abstract syntax tree** (**AST**) is a data structure used in computer science to represent the structure of a program or code snippet. It is a tree representation of the abstract syntactic structure of text (often source code) written in a formal language. Each node of the tree denotes a construct occurring in the text. It is sometimes called just a **syntax tree**.

The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. For instance, grouping parentheses are implicit in the tree structure, so these do not have to be represented as separate nodes. Likewise, a syntactic construct like an if-condition-then statement may be denoted by means of a single node with three branches.

This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees. Parse trees are typically built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing, e.g., contextual analysis.

Abstract syntax trees are also used in program analysis and program transformation systems.



An abstract syntax tree for the following code for the Euclidean algorithm:

```
while b ≠ 0:
    if a > b:
        a := a − b
    else:
        b := b − a
return a
```

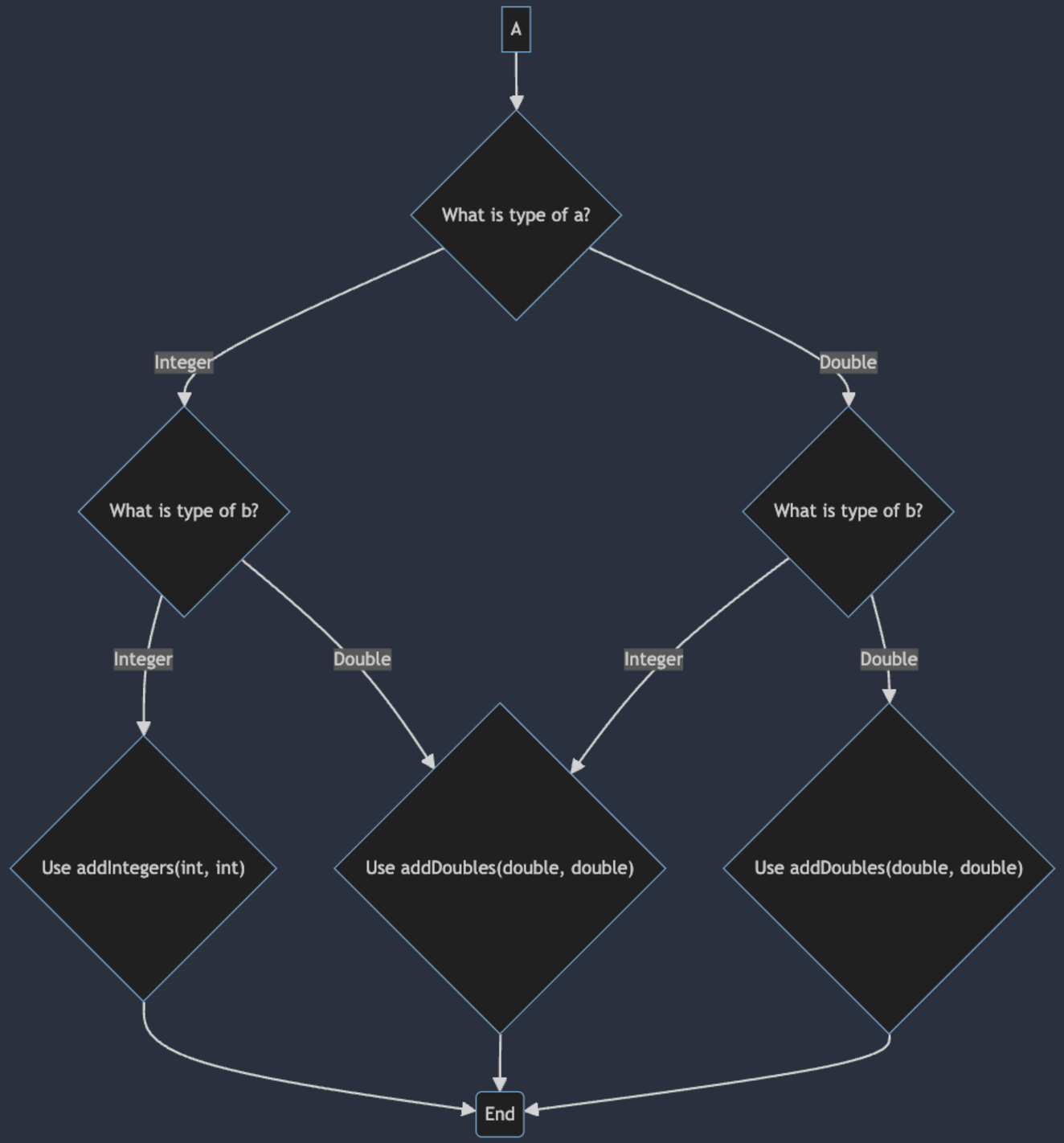
Application in compilers [edit]

Abstract syntax trees are data structures widely used in compilers to represent the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. It often serves as an intermediate representation of the program through several stages that the compiler requires, and has a strong impact on the final output of the compiler.

Motivation [edit]

An AST has several properties that aid the further steps of the compilation process:

```
function add(a, b) {
    return a + b;
}
```


A
What is type of a?
Integer
Double
What is type of b?
What is type of b?
Integer
Double
Integer
Double
Use addIntegers(int, int)
Use addDoubles(double, double)
Use addDoubles(double, double)
End

JS

Self-Optimizing AST Interpreters

Thomas Würthinger* Andreas Wöß† Lukas Stadler† Gilles Duboscq†
Doug Simon* Christian Wimmer*

*Oracle Labs †Institute for System Software, Johannes Kepler University Linz, Austria
thomas.wuerthinger@oracle.com woess@ssw.jku.at stadler@ssw.jku.at duboscq@ssw.jku.at
doug.simon@oracle.com christian.wimmer@oracle.com

Abstract

An abstract syntax tree (AST) interpreter is a simple and natural way to implement a programming language. However, it is also considered the slowest approach because of the high overhead of virtual method dispatch. Language implementers therefore define bytecodes to speed up interpretation, at the cost of introducing inflexible and hard to maintain bytecode formats. We present a novel approach to implementing AST interpreters in which the AST is modified during interpretation to incorporate type feedback. This tree rewriting is a general and powerful mechanism to optimize many constructs common in dynamic programming languages. Our system is implemented in Java™ and uses the static typing and primitive data types of Java elegantly to avoid the cost of boxed representations of primitive values in dynamic programming languages.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments, Optimization

General Terms Algorithms, Languages, Performance

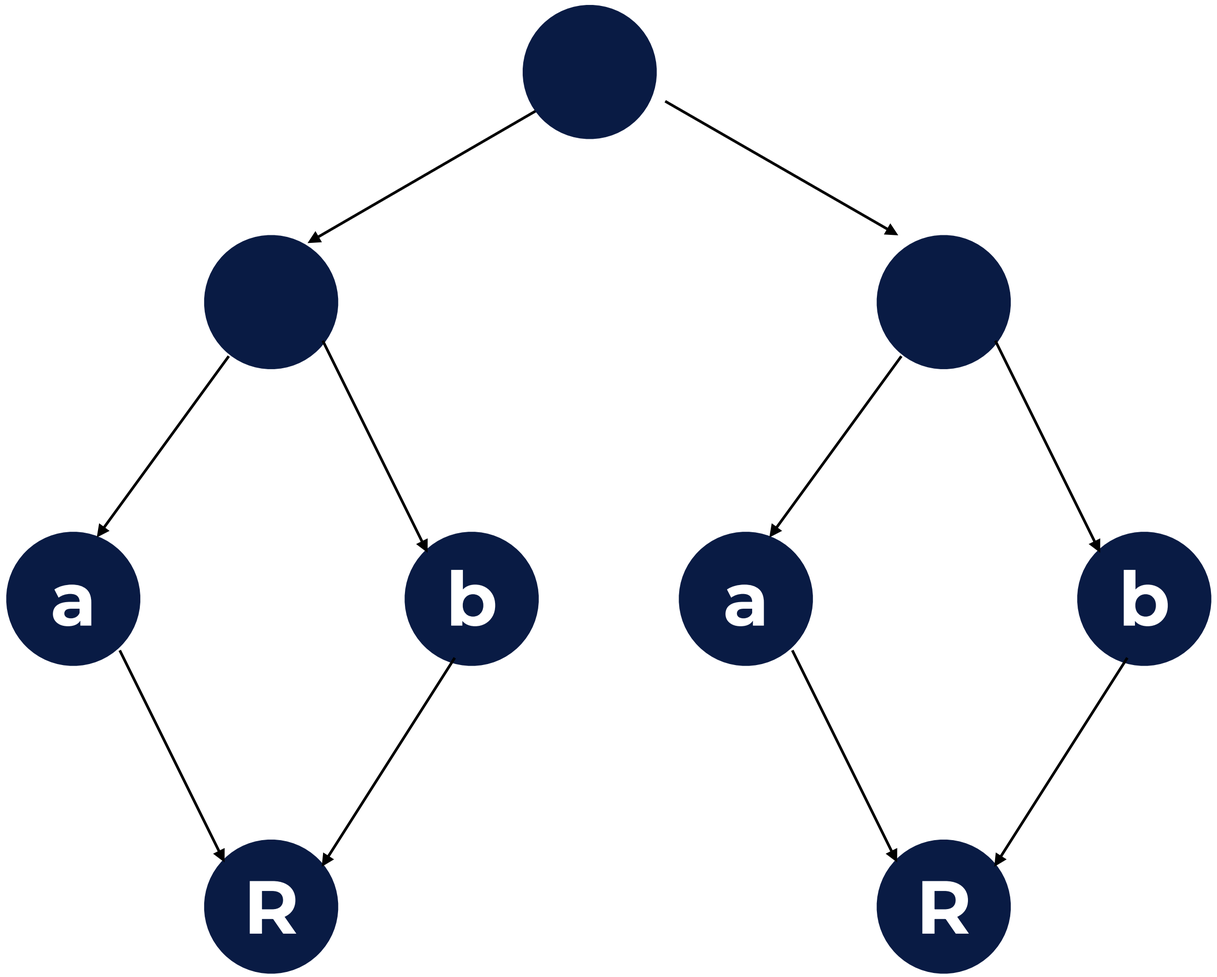
Keywords Java, JavaScript, dynamic languages, virtual machine, language implementation, optimization

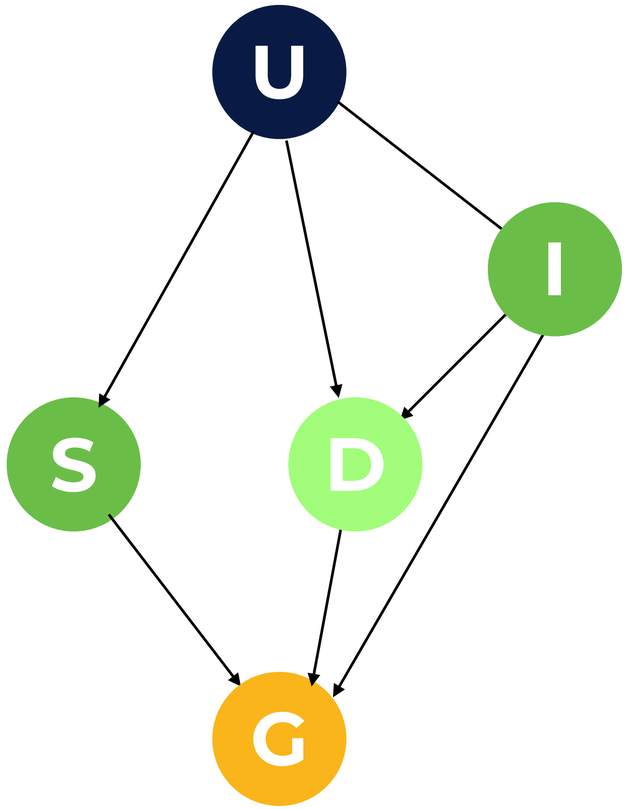
studied [9, 22]. Bytecodes are a virtual instruction set, i.e, they can be defined freely by language implementers without worrying about hardware constraints. There are two different strategies when defining bytecodes. We believe that both are cumbersome when implementing a new language:

- Language-independent bytecodes aim to provide an instruction set that is suitable for many languages. Java™ bytecodes [17], originally designed primarily for the Java programming language, are now used for many languages. The Common Intermediate Language (CIL) [8], which is used by the .NET system, was specifically designed for multiple languages. Language-independent bytecodes provide only general instructions whose semantics do not match the language semantics exactly. Long bytecode sequences are potentially required to achieve the desired language behavior. For example, Java bytecodes are statically typed, so compiling dynamically typed JavaScript requires frequent type checks. A simple addition in JavaScript requires checking whether the operands are numbers, strings, or arbitrary objects, which all require different handling. Therefore, implementations of JavaScript that run on a Java VM often call a method for a seemingly simple addition.

PyPy (/ˈpaɪpaɪ/) is an implementation of the Python programming language.[2] PyPy often runs faster than the standard implementation CPython because PyPy uses a just-in-time compiler.[3] Most Python code runs well on PyPy except for code that depends on CPython extensions, which either does not work or incurs some overhead when run in PyPy.

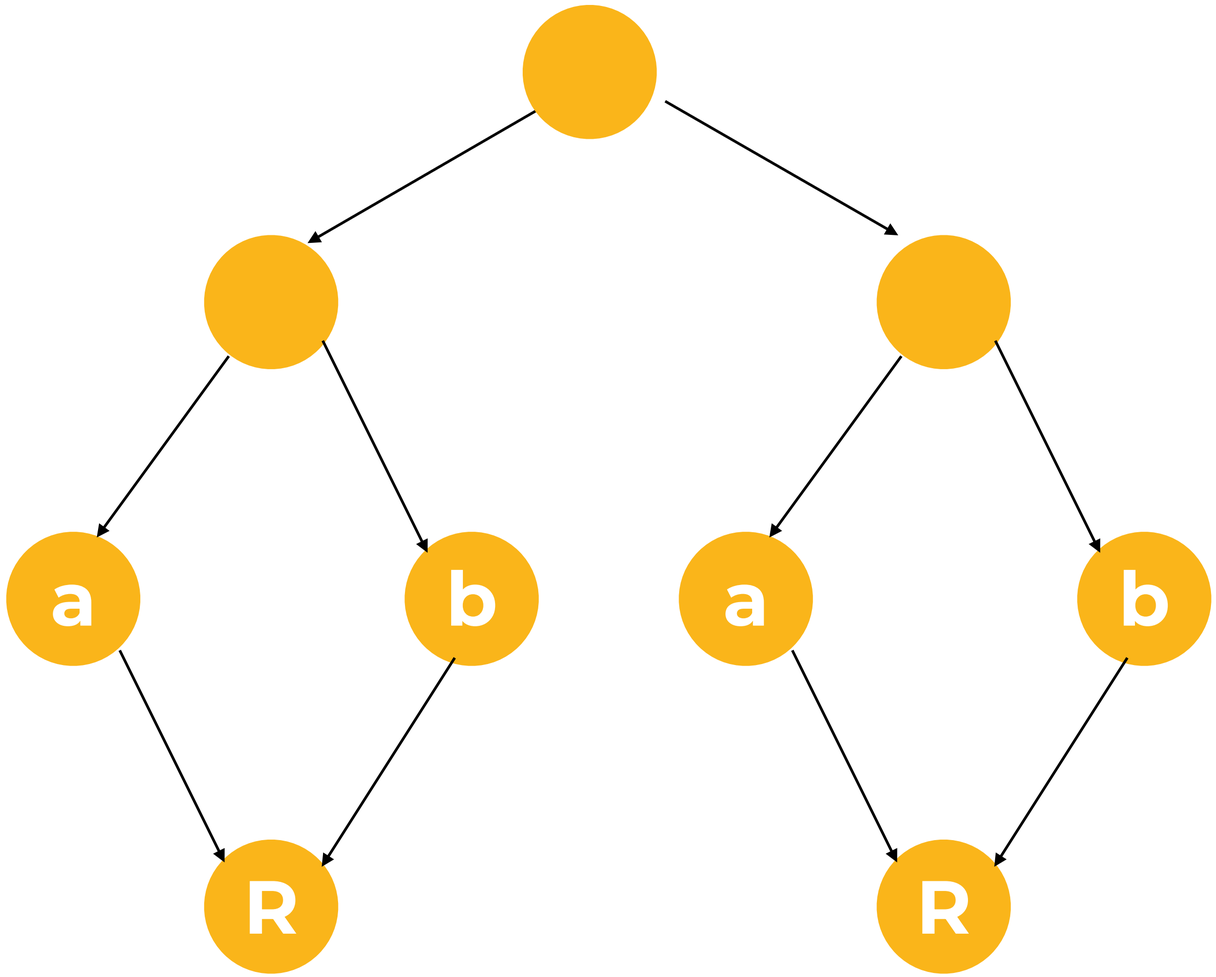
PyPy itself is built using a technique known as meta-tracing, which is a mostly automatic transformation that takes an interpreter as input and produces a tracing just-in-time compiler as output. Since interpreters are usually easier to write than compilers, but run slower, this technique can make it easier to produce efficient implementations of programming languages. PyPy's meta-tracing toolchain is called RPython.

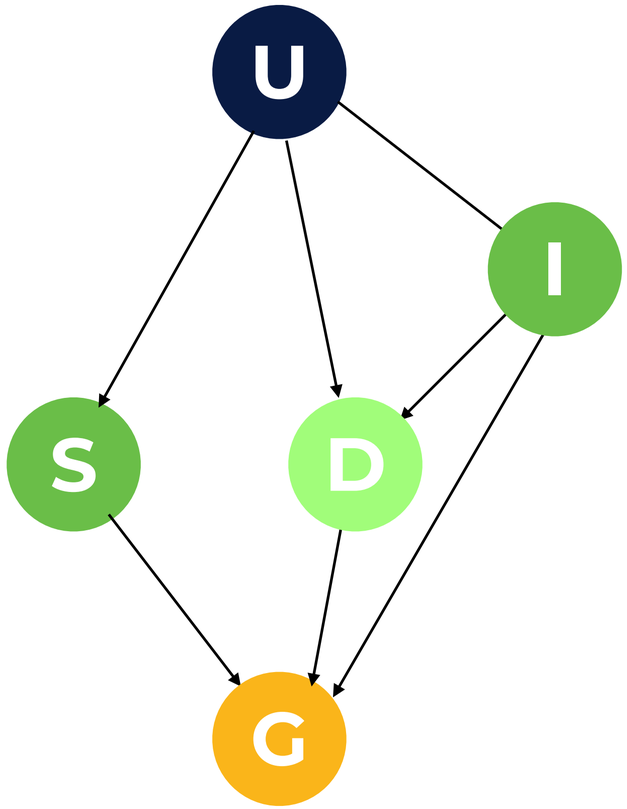




Tranzycje

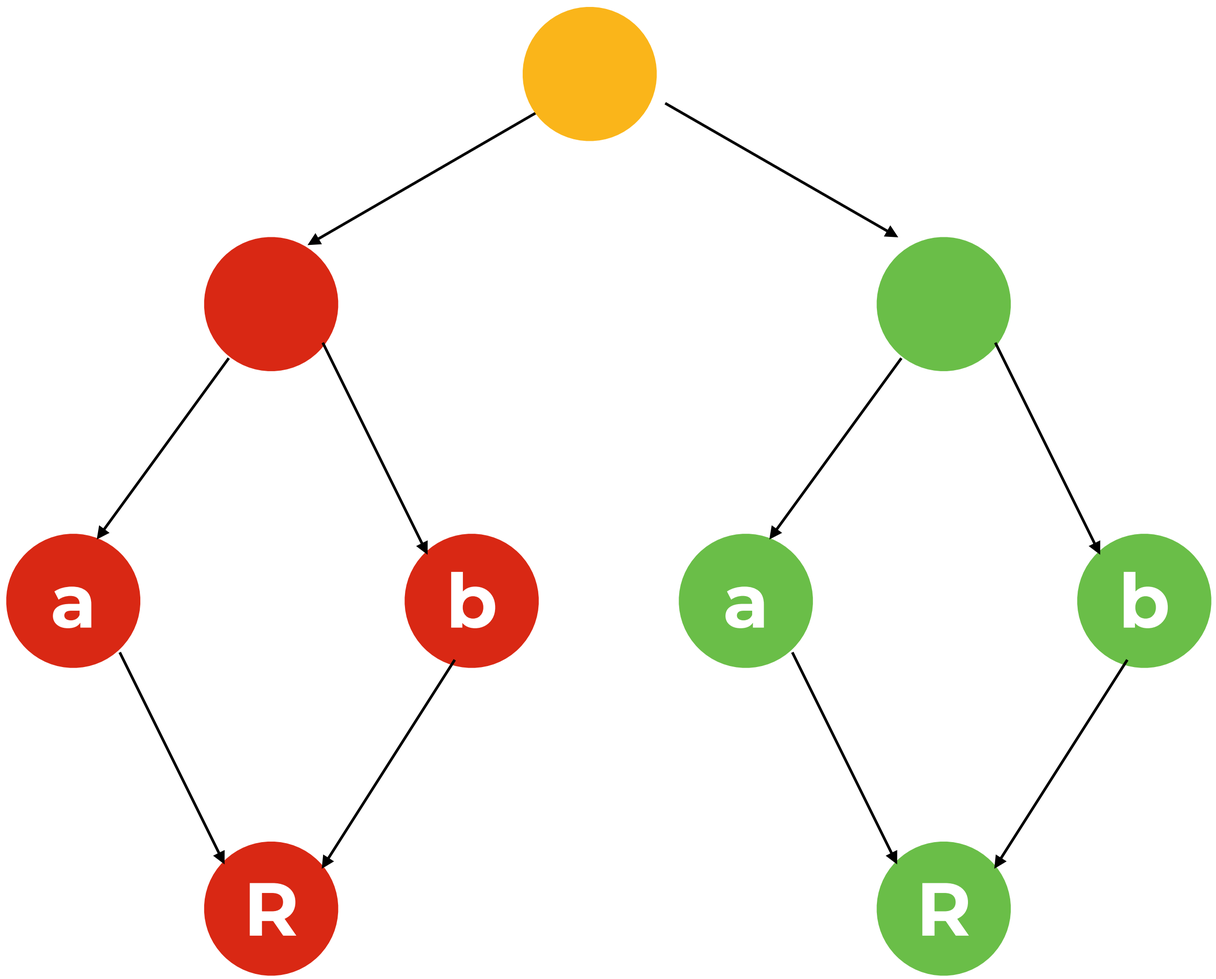
AST z niezainicjowanymi Node

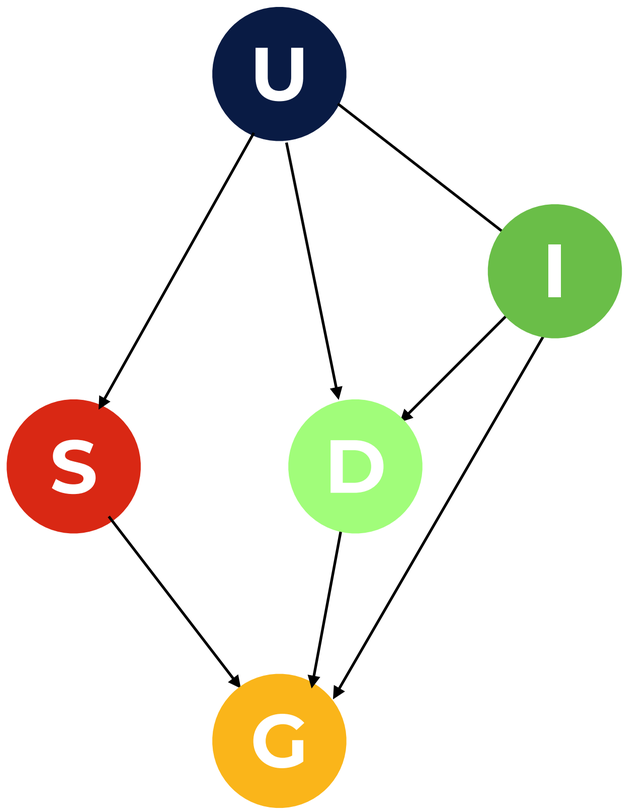




Tranzycje

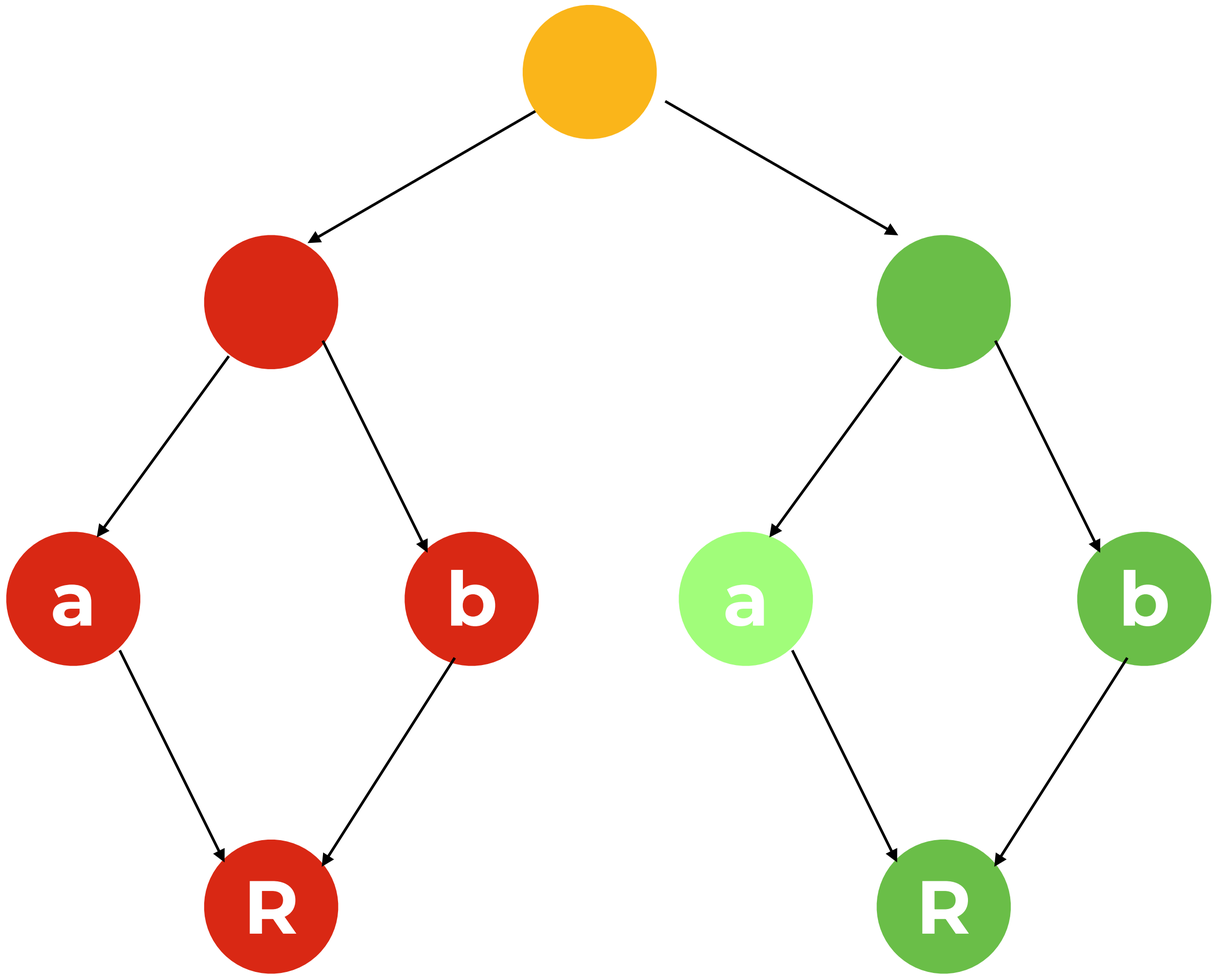
Profiling AST

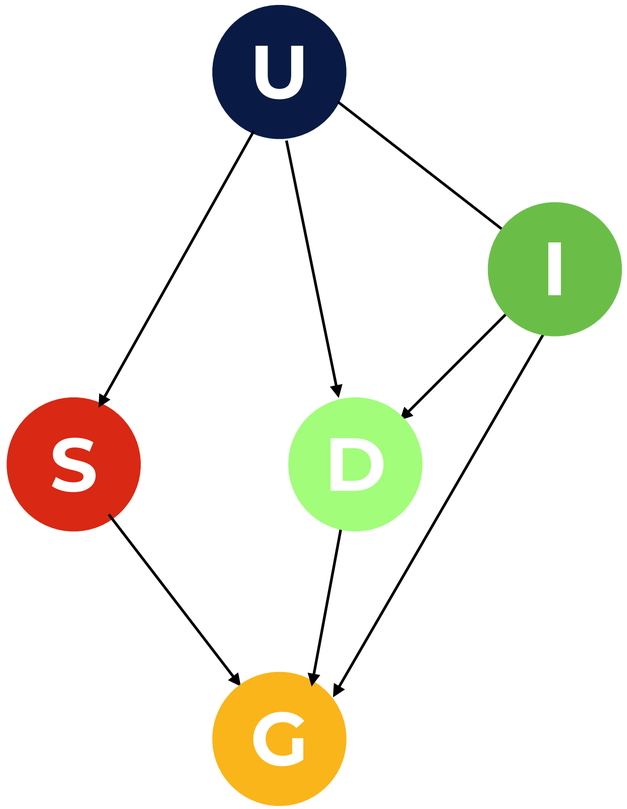




Tranzycje

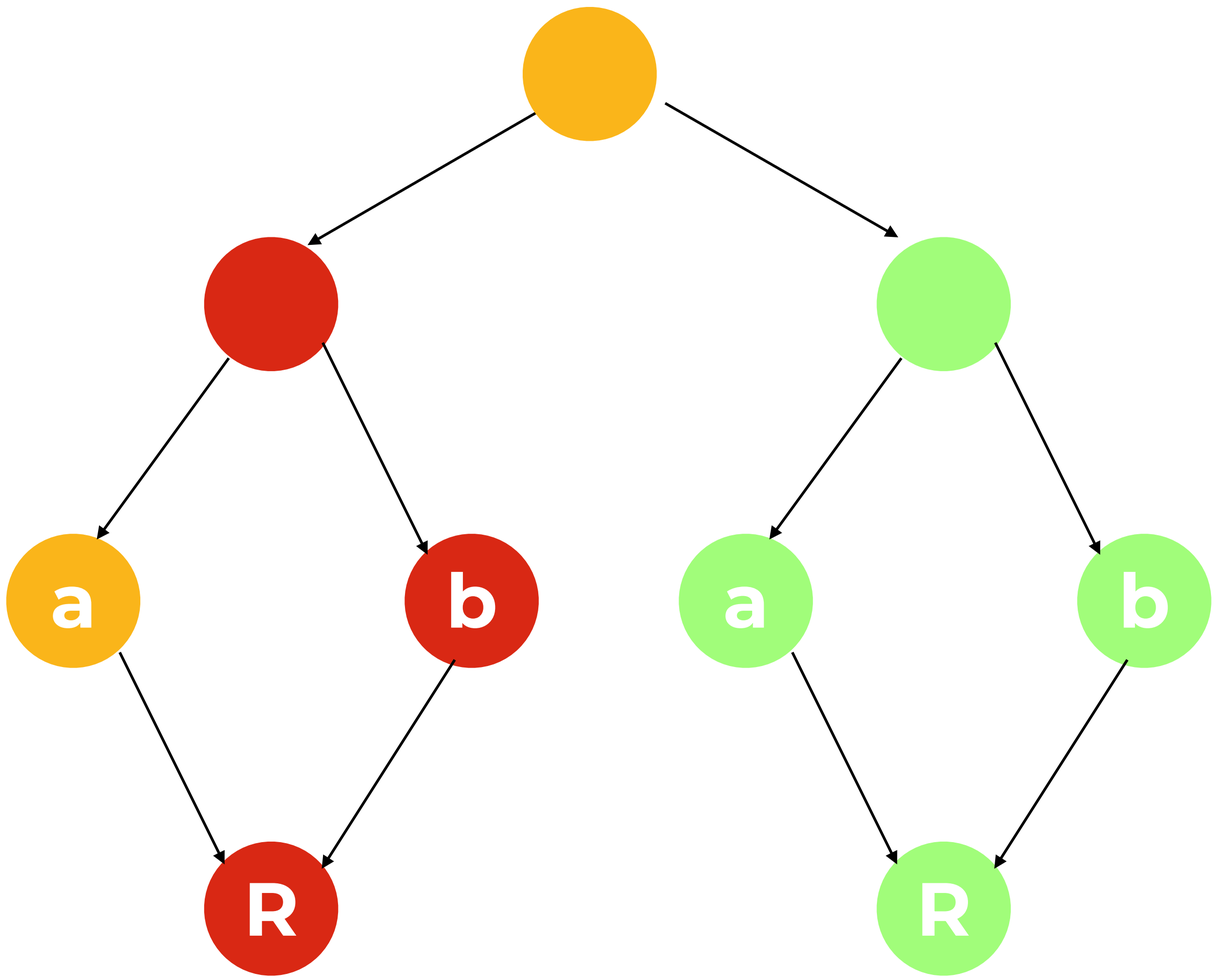
Profiling AST

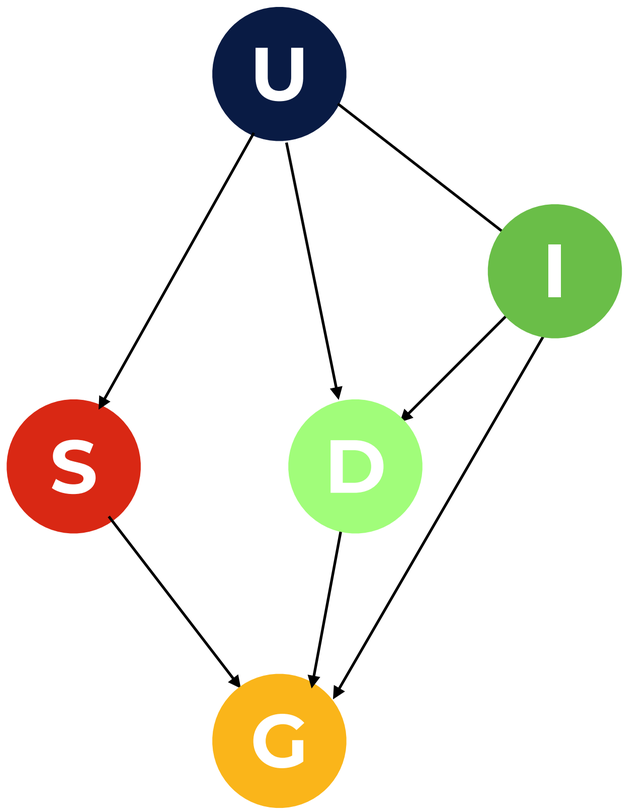




Tranzycje

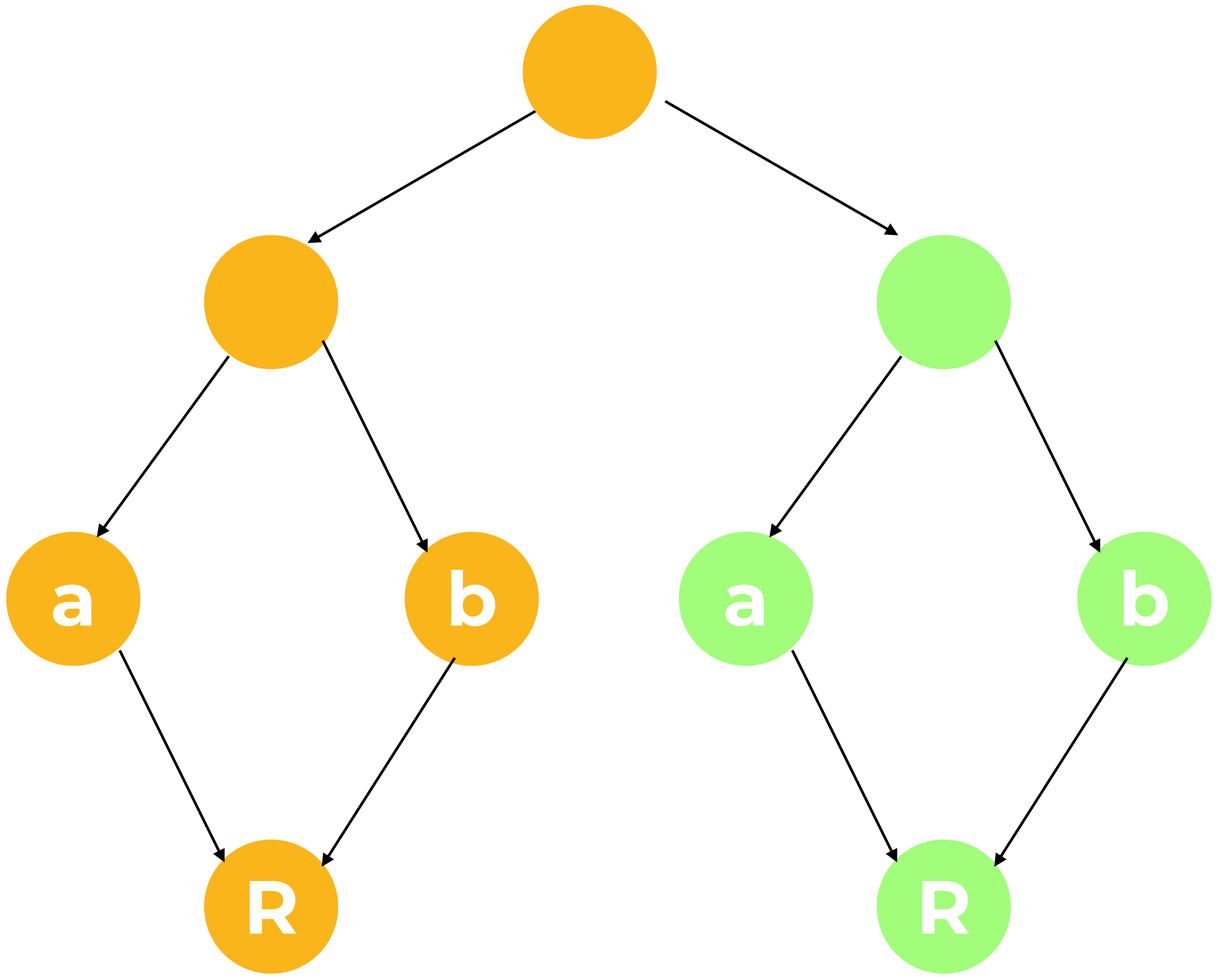
Profiling AST

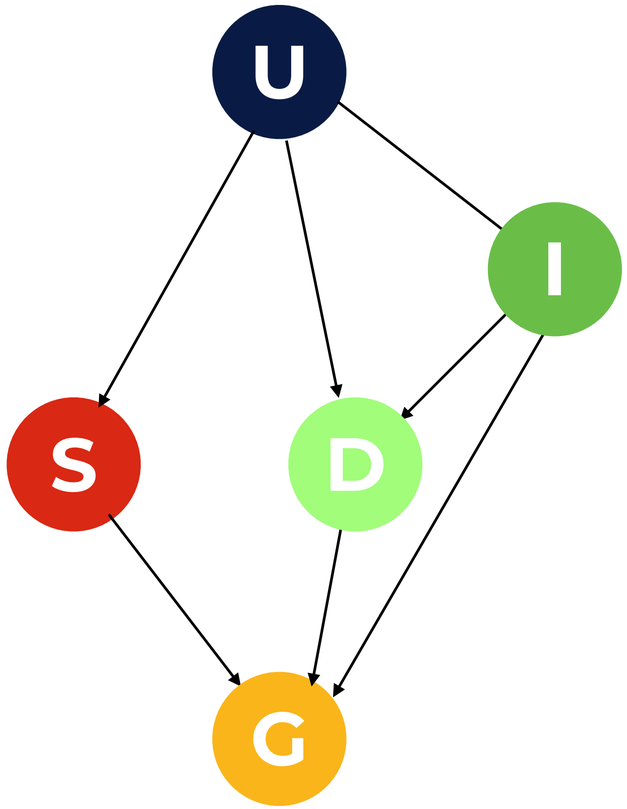




Tranzycje

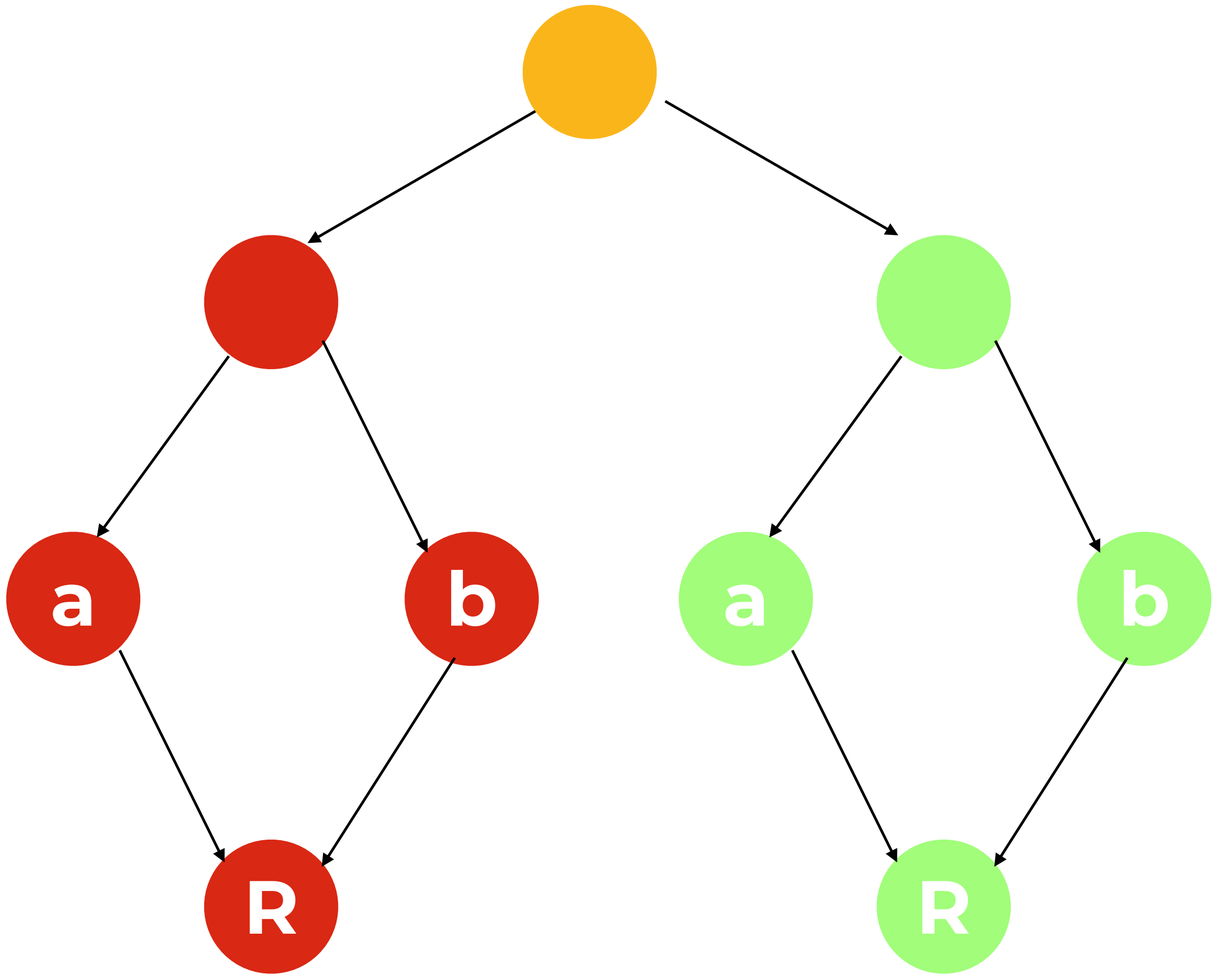
Profiling AST

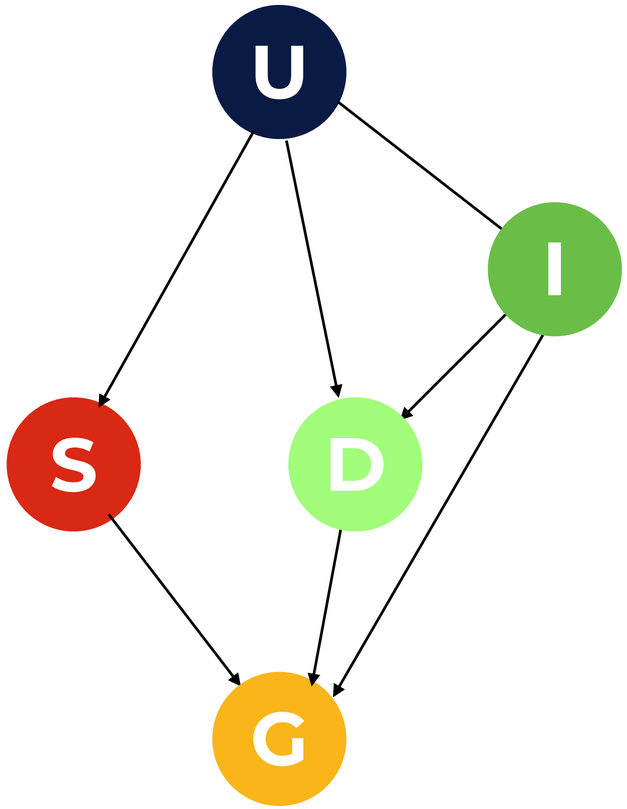




Tranzycje

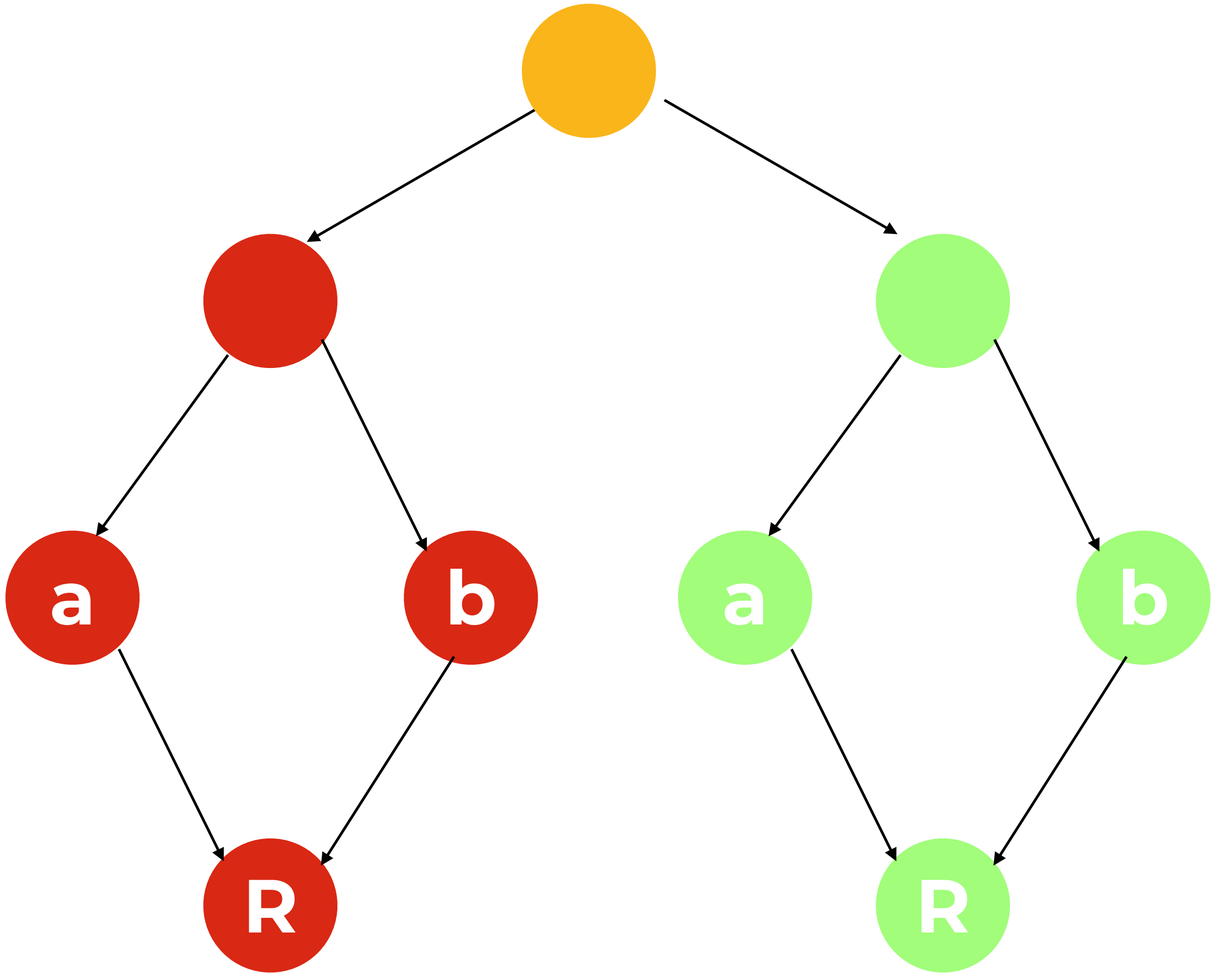
Profiling AST





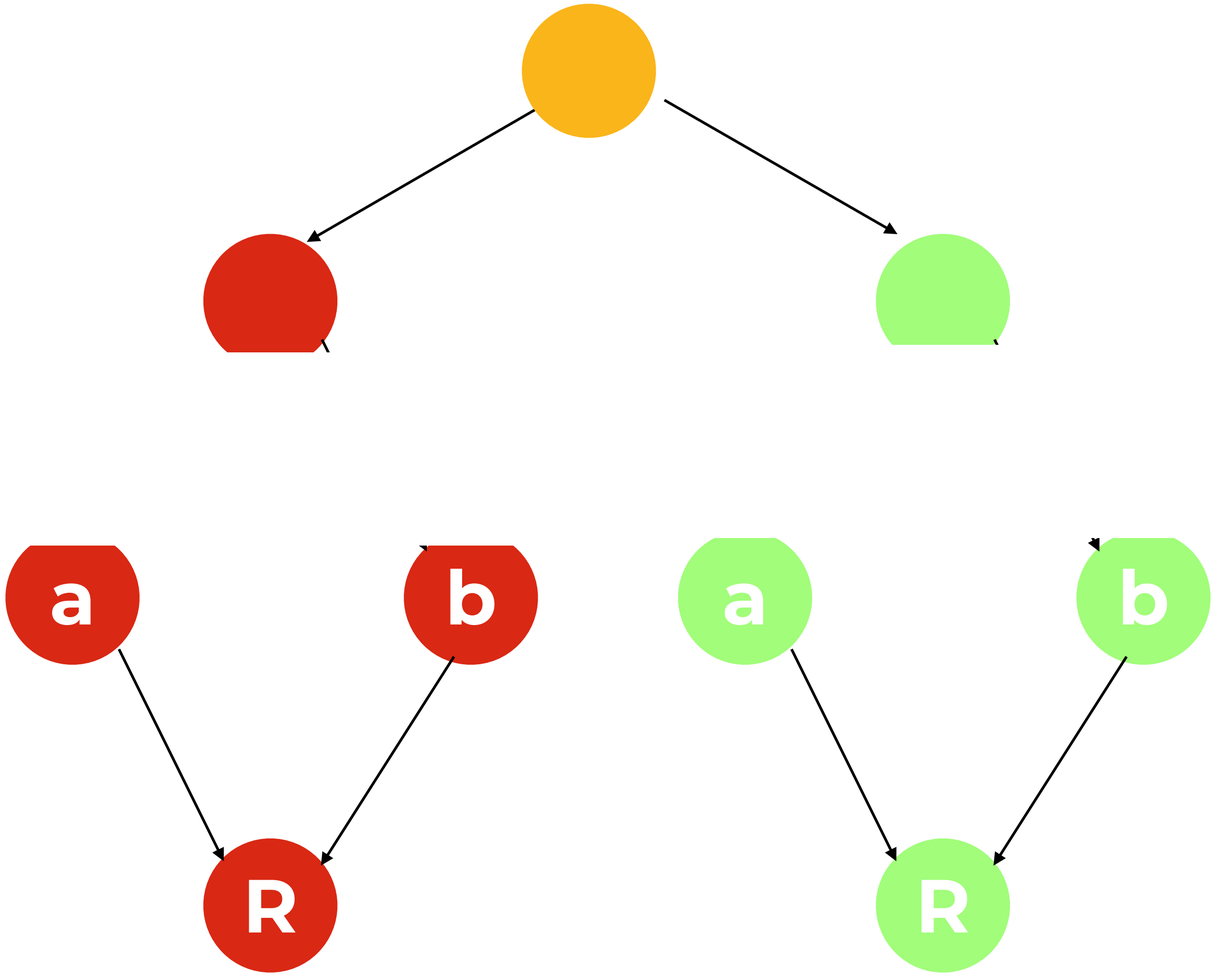
Tranzycje

Profiling AST

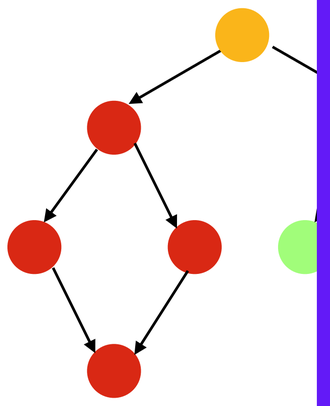


Ustabilizowane, monformiczne AST

Once again - speculative optimization

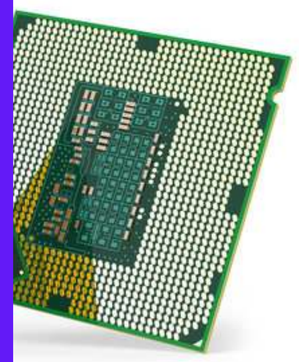


Guard Nodes



Partial evaluation is a key optimization technique where the Truffle framework precomputes parts of the program based on known inputs and constant values at compile time.

This results in highly optimized machine code for guest languages, significantly improving the performance of language interpreters built with Truffle.



Partial Evaluation

Compilers for free with weval

May 19, 2024

Chris Fallin came and gave a talk to the Northeastern Programming Research Laboratory last month. He talked about his work on a new project called weval, a WebAssembly partial evaluator (and then helped me write this post!).

Partial evaluation is neat. In short, it's all about taking an existing program, modifying it to hold some of its inputs as constants, and then letting the compiler/optimizer go hog wild on it. The result is still a program—not a value—and it's usually faster than the original program.

The usual small example is the power function. If you have a function that takes two arguments, `x` and `y`, and returns `x^y`:

```
int power(int x, int y) {
  int result = 1;
  for (int i = 0; i < y; i++) {
    result *= x;
  }
  return result;
}
```

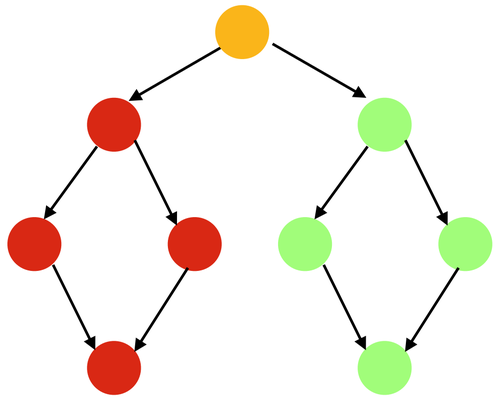
If you partially evaluate this function with respect to `y` at `y = 5`, you get a new function that takes one argument, `x`, and returns `x^5`:

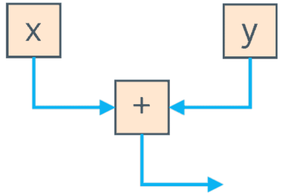
```
int power_5(int x) {
  int result = 1;
  for (int i = 0; i < 5; i++) {
    result *= x;
  }
  return result;
}
```

Now, to you, this might not look that different from the original function. But to an optimizer, it is a new world of opportunity. The optimizer can unroll the loop and remove the conditional:

```
int power_5(int x) {
  return x * x * x * x * x;
}
```

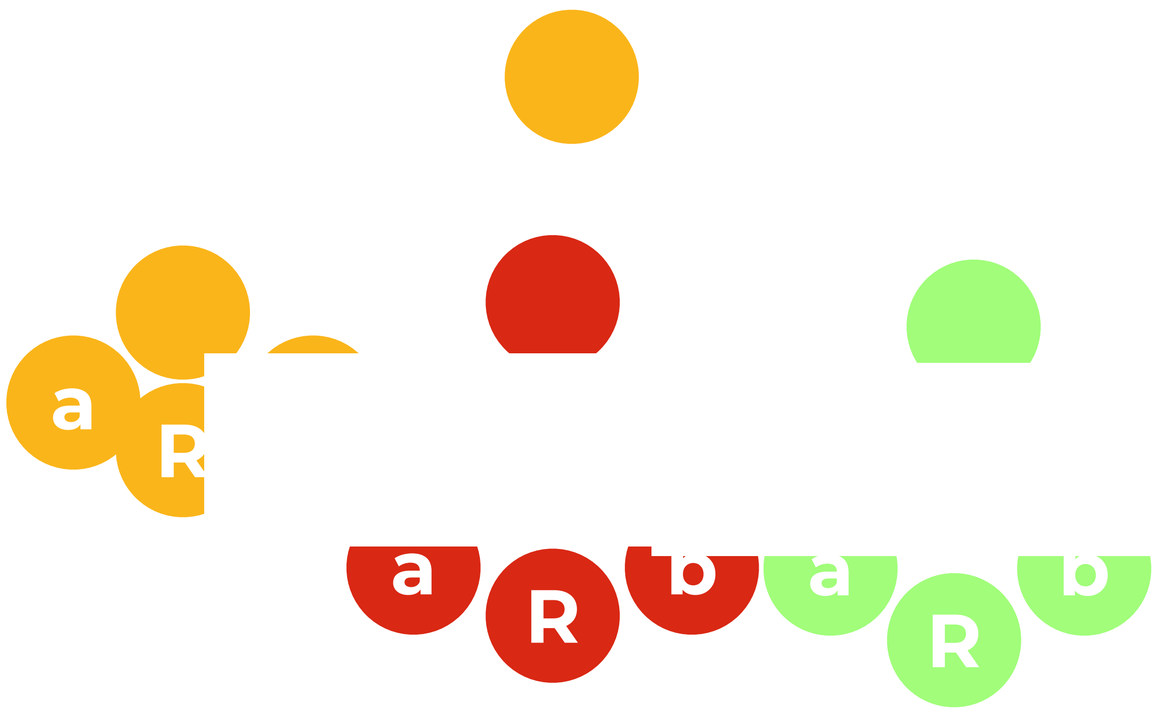
Partial Evaluation



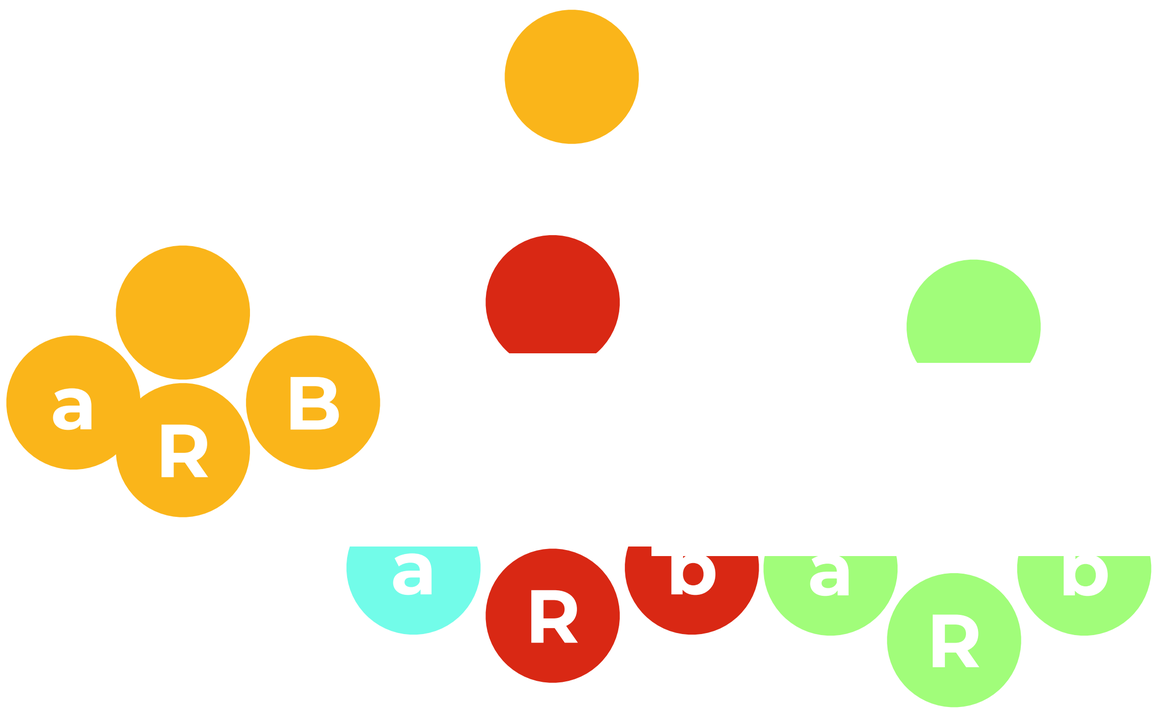


Graal Compiler

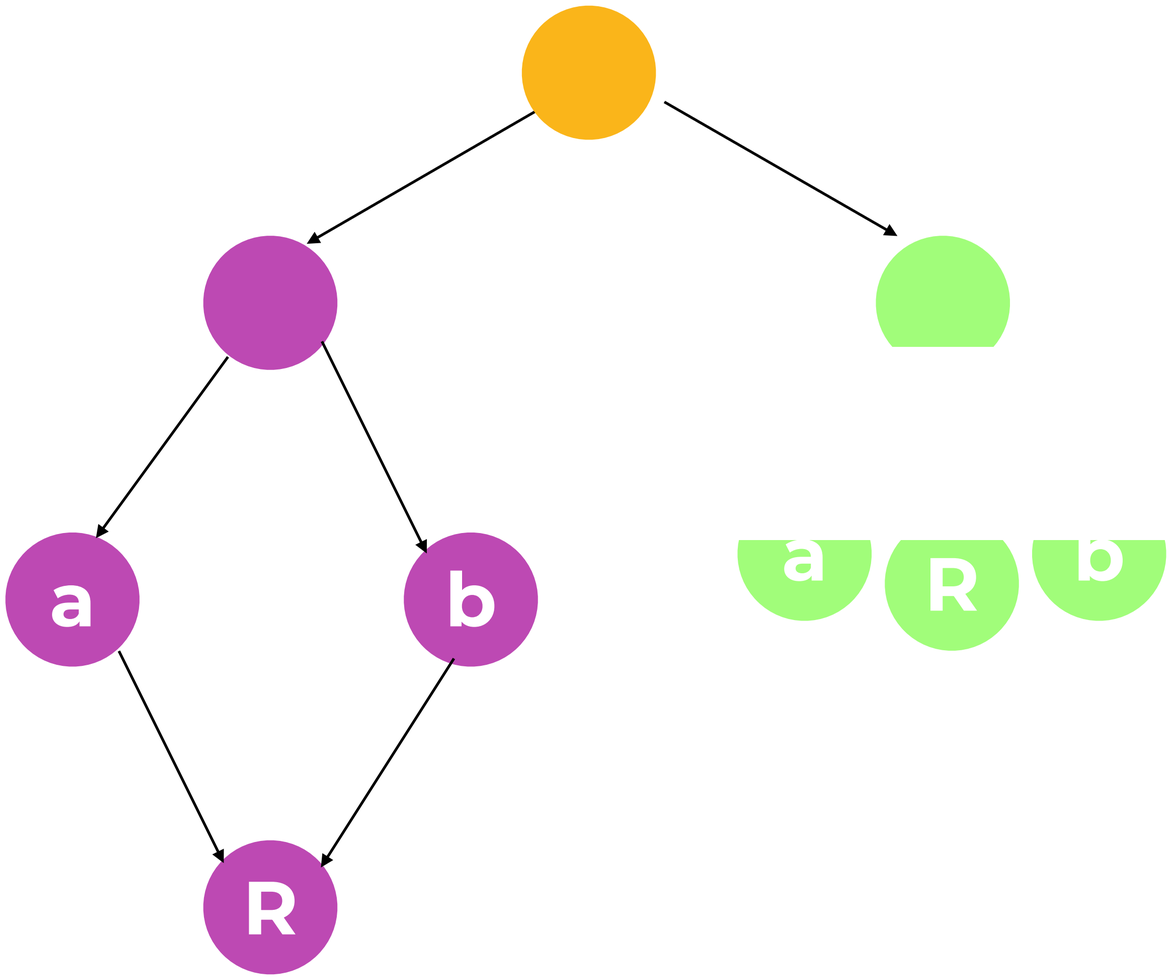
Partial Evaluation

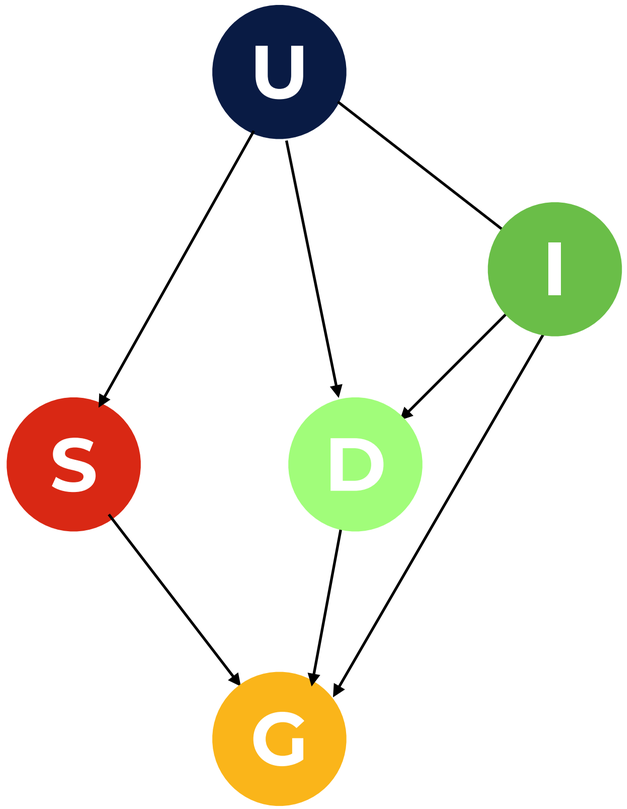


Partial Evaluation



Partial Deoptimization & Reprofiling





Tranzycje

Partial Deoptimization & Reprofiling

Alright, so how do we put this into practice?

Language Interpreter

GraalVM in Truffle Mode

Truffle Unchained — Portable Language Runtimes as Java Libraries

Christian Humer · Follow
Published in graalvm · 7 min read · Sep 19, 2023

78

For the past couple of years, Truffle languages were installed using our built-in GraalVM Updater tool called `gu`. To install a language, you downloaded Oracle GraalVM and then ran the `gu install` command to download and unpack components. Together with the GraalVM for JDK 21 and Truffle/Polyglot 23.1 releases, we remove `gu` and provide languages as Maven dependencies instead. In this blog post, we want to dive into why this change was necessary, what we gain from it, and what you need to do to

GraalVM as library

```
dependencies {
    implementation("org.graalvm.polyglot:polyglot:23.1.0")
    // Select language: js, ruby, python, java, llvm, wasm, languages
    implementation("org.graalvm.polyglot:js:23.1.0")
}
```

Graaljs » 23.0.3

Graal JavaScript engine

License	MIT UPL
Categories	JavaScript Processors
Tags	graalvm javascript processor
HomePage	http://www.graalvm.org/
Date	Jan 17, 2024
Files	pom (1 KB) jar (26.1 MB) View All
Repositories	Central
Ranking	**#1943 in MvnRepository (See Top Artifacts)** **#3 in JavaScript Processors**
Used By	**243 artifacts**

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

```
<!-- https://mvnrepository.com/artifact/org.graalvm.js/js -->
<dependency>
    <groupId>org.graalvm.js</groupId>
    <artifactId>js</artifactId>
    <version>23.0.3</version>
</dependency>
```

☑ Include comment with link to declaration

```
try (Context context = Context.create()) {
    Value result = context.eval("js",
       "({ " +
        "id : 42," +
        "text : '42', " +
        "arr : [1,42,3] " +
        "}))");

    assert result.hasMembers();

    int id = result.getMember("id").asInt();
    assert id == 42;

    String text = result.getMember("text").asString();
    assert text.equals("42");

    Value array = result.getMember("arr");

    assert array.hasArrayElements();

    assert array.getArraySize() == 3;

    assert array.getArrayElement(1).asInt() == 42;
}
```

GraalVM as library

```
dependencies {
    implementation("org.graalvm.polyglot:polyglot:23.1.0")
    // Select language: js, ruby, python, java, llvm, wasm, languages
    implementation("org.graalvm.polyglot:js:23.1.0")
}
```

Graaljs » 23.0.3

Graal JavaScript engine

License	MIT UPL
Categories	JavaScript Processors
Tags	graalvm javascript processor
HomePage	http://www.graalvm.org/
Date	Jan 17, 2024
Files	pom (1 KB) jar (26.1 MB) View All
Repositories	Central
Ranking	#1943 in MvnRepository (See Top Artifacts) #3 in JavaScript Processors
Used By	243 artifacts

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

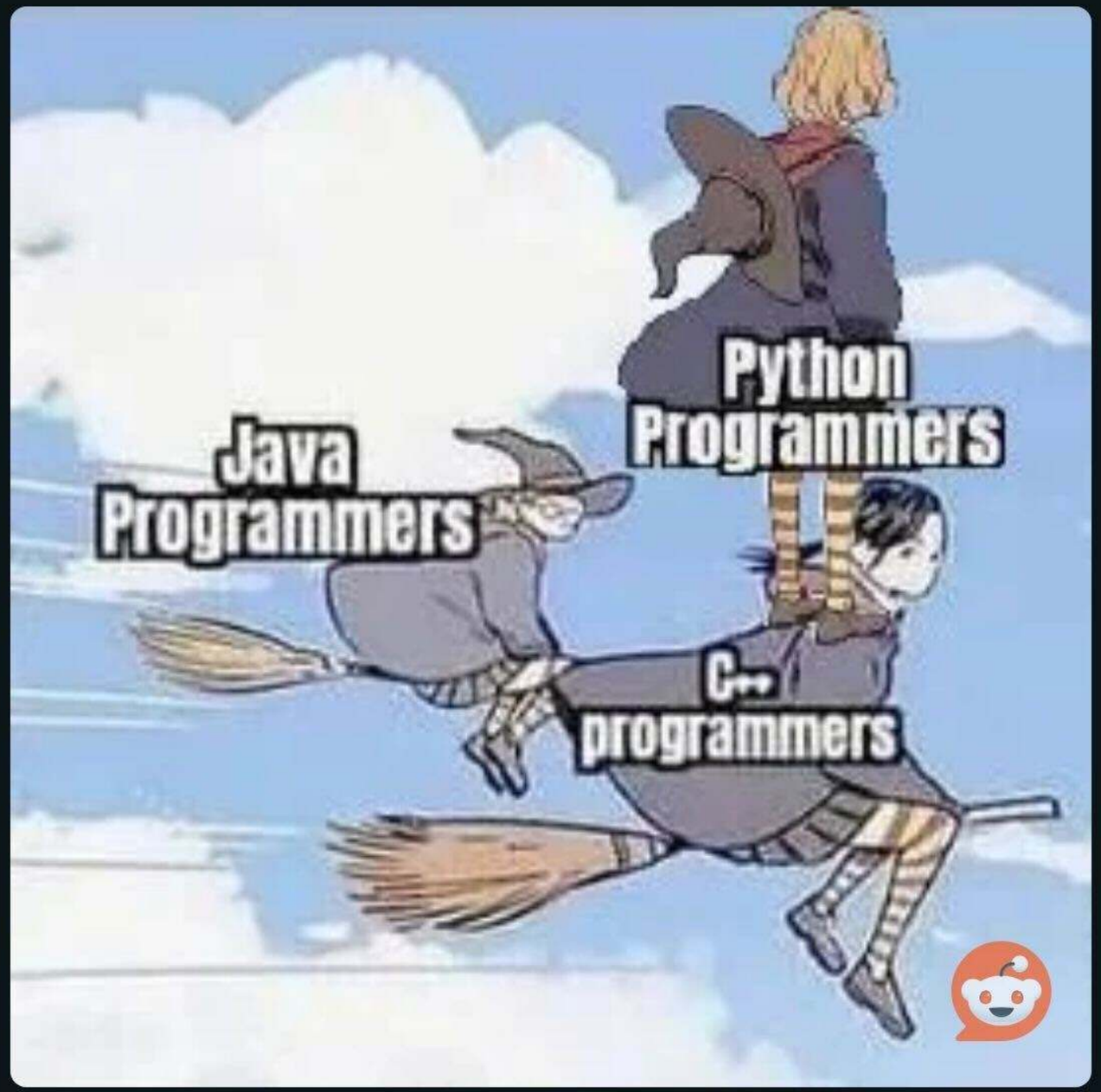
```
<!-- https://mvnrepository.com/artifact/org.graalvm.js/js -->
<dependency>
    <groupId>org.graalvm.js</groupId>
    <artifactId>js</artifactId>
    <version>23.0.3</version>
</dependency>
```

☑ Include comment with link to declaration

```
public static class MyClass {

    public int id = 42;
    public String text = "42";
    public int[] arr = new int[]{1, 42, 3};

    public Callable<Integer> ret42 = () -> 42;

    public static void main(String[] args) {
        try (Context context = Context.newBuilder()
        .allowAllAccess(true)
        .build()) {
            context.getBindings("js").putMember(
              "javaObj", new MyClass());

            boolean valid = context.eval("js",

            " javaObj.id == 42" +
            " && javaObj.text == '42'" +
            " && javaObj.arr[1] == 42" +
            " && javaObj.ret42() == 42")
            .asBoolean();
            assert valid == true;
        }
    }
}
```

GraalVM as library

From **ProgrammerHumor** community on **Reddit**



JVM
JVM WHEN YOU REALIZE IT'S WRITTEN IN C++
imgflip.com

Maxine Virtual Machine

1 language

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

This article has multiple issues. Please help **improve it** or discuss these issues on the **talk page**. *(Learn how and when to remove these template messages)* [hide]

- This article includes a list of general references, but **it lacks sufficient corresponding inline citations**. *(August 2012)*
- This article **relies excessively on references to primary sources**. *(June 2011)*

The **Maxine** virtual machine is an open source virtual machine that is developed at the University of Manchester.[2] It was formerly developed by Sun Microsystems Laboratories,[3] since renamed Oracle Labs. The emphasis in Maxine's software architecture is on modular design and code reuse for flexibility, configurability, and productivity for industrial and academic virtual machine researchers. It is one of a growing number of Java virtual machines written entirely in Java in a meta-circular style. Examples include Squawk and Jikes RVM.

Maxine

Original author(s)	Bernd Mathiske, Douglas Simon (Sun Labs, Oracle)
Developer(s)	University of Manchester, Maxine team
Initial release	2005; 19 years ago
Stable release	2.9.0[1] / December 20, 2019; 4 years ago
Repository	github.com/beehive-lab/Maxine-VM
Written in	Java
Operating system	Solaris, Linux, macOS
Platform	x64, ARM32, AArch64, RISC-V
Type	Java virtual machine
License	GPL version 2.0
Website	github.com/beehive-lab/Maxine-VM

Architecture [edit]

The Maxine VM is characterized internally by aggressive use of advanced language features in Java 1.5 and 1.6, by modular subsystems coordinated through Java interfaces, by the absence of an interpreter, and by a tightly coupled debugger and visualization tool for VM development.

Compatibility [edit]

Maxine is plug compatible with an unmodified Java Development Kit (JDK). Maxine can be developed, built, and run in standard Java integrated development environments (IDEs), including NetBeans, Eclipse, and IntelliJ IDEA.

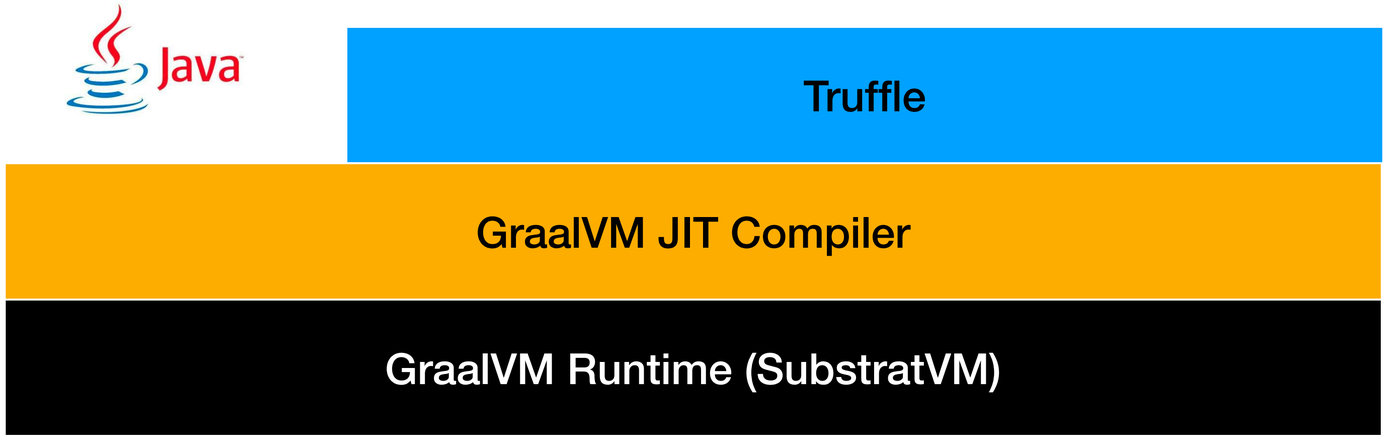
Systems programming in Java [edit]

GraalVM Runtime (SubstratVM)

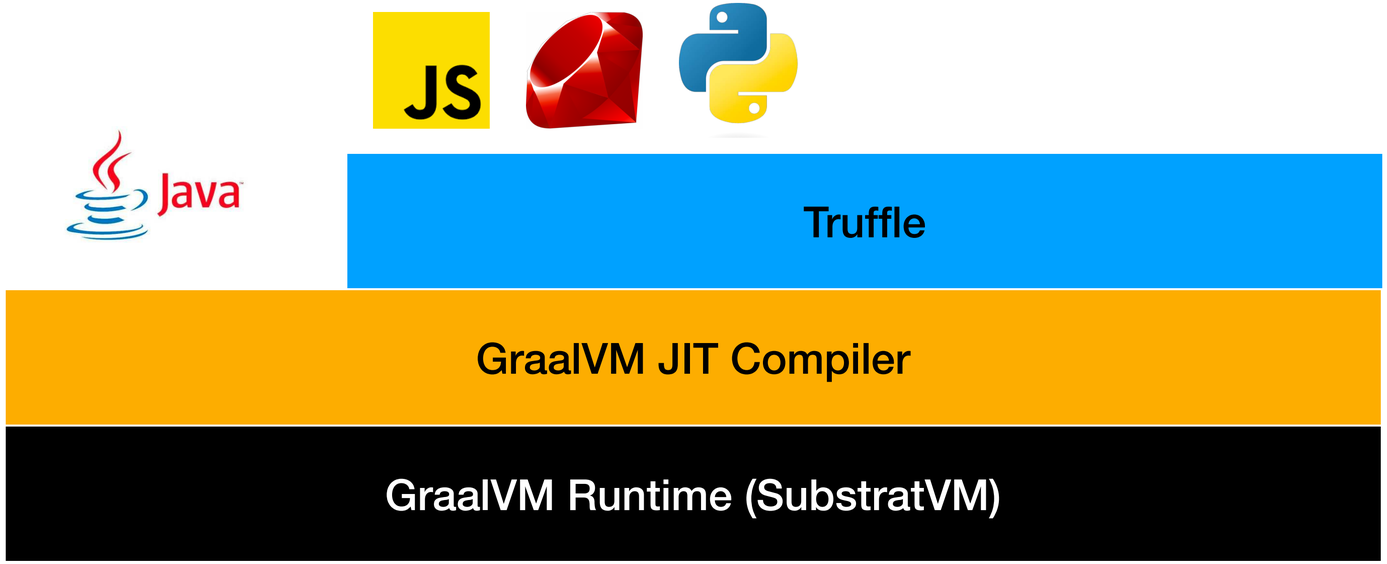
GraalVM JIT Compiler

GraalVM Runtime (SubstratVM)

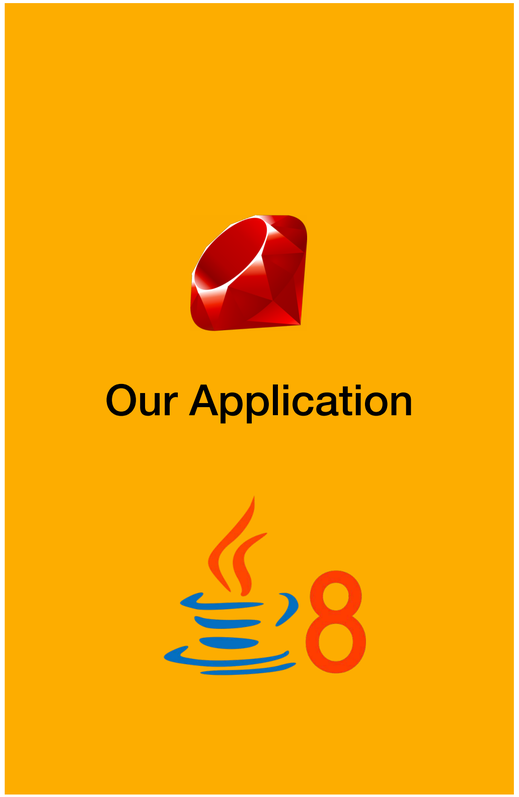
espresso

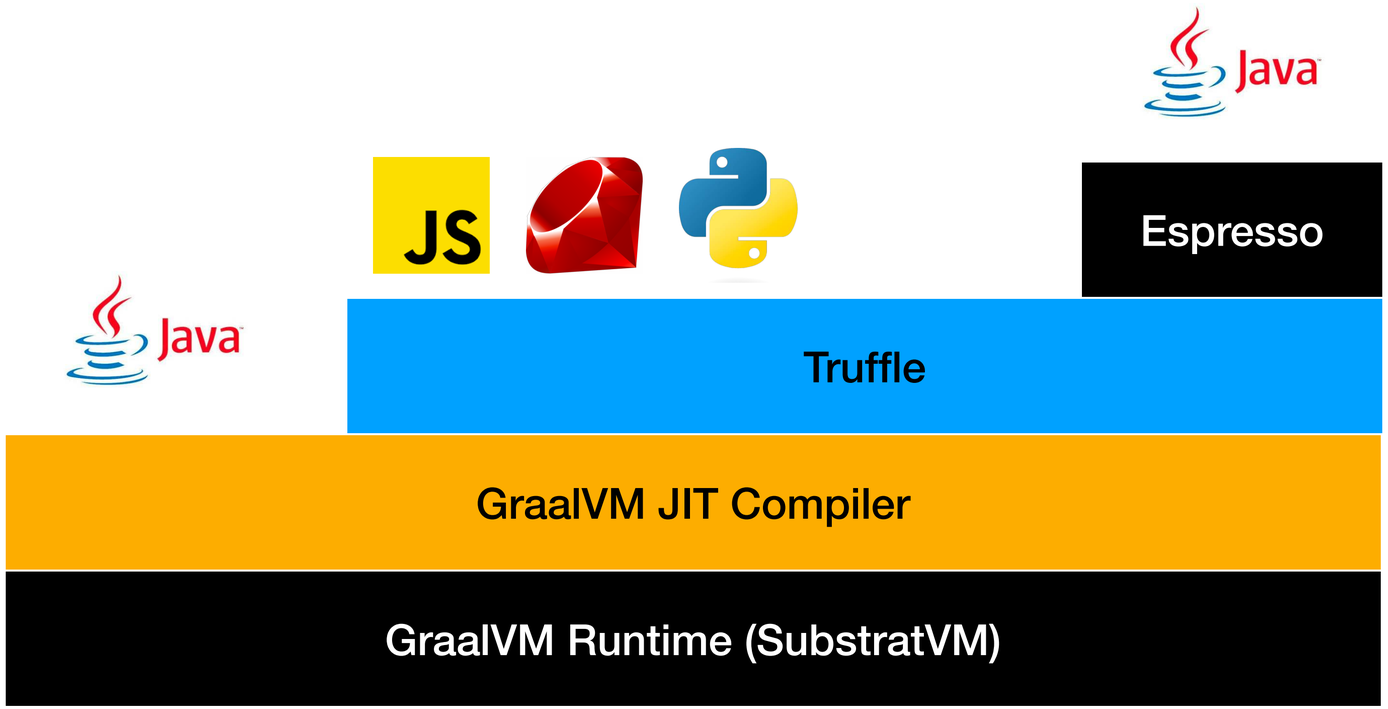

Java
Truffle
GraalVM JIT Compiler
GraalVM Runtime (SubstratVM)

espresso

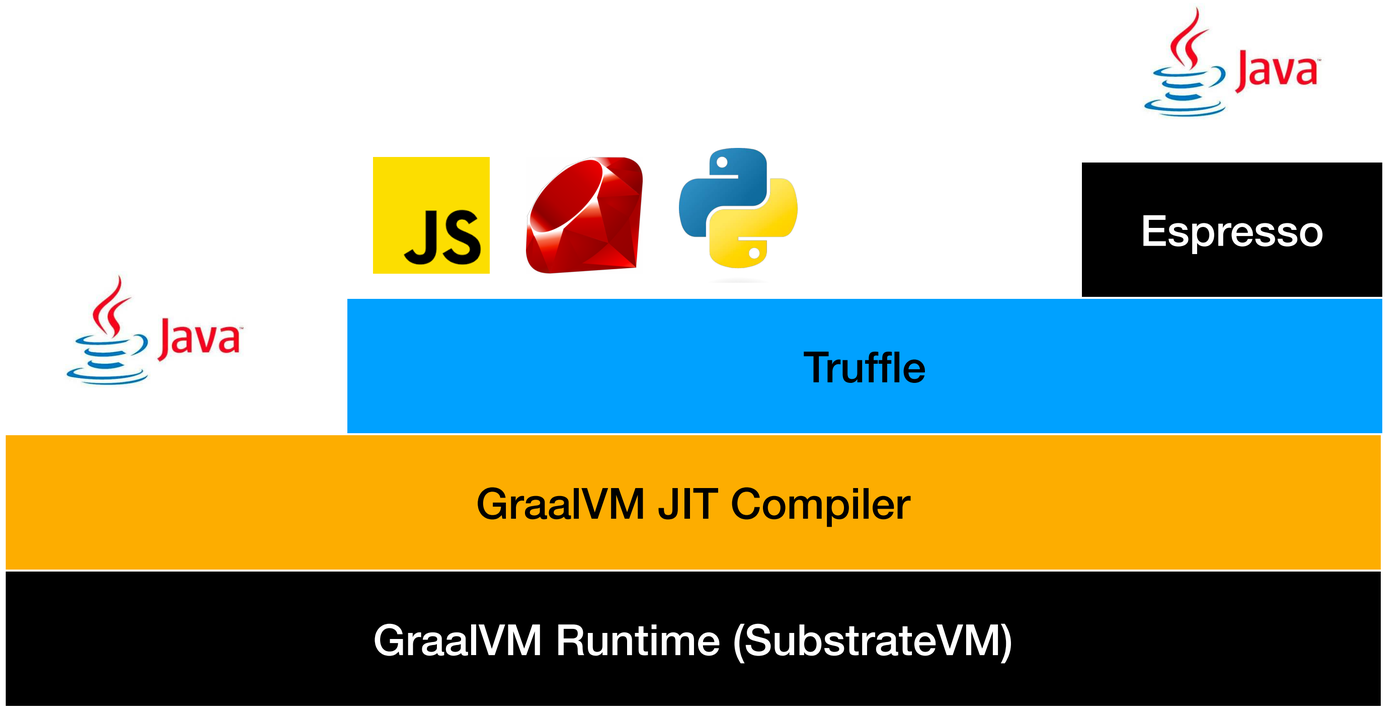

JS
Java
Truffle
GraalVM JIT Compiler
GraalVM Runtime (SubstratVM)

espresso

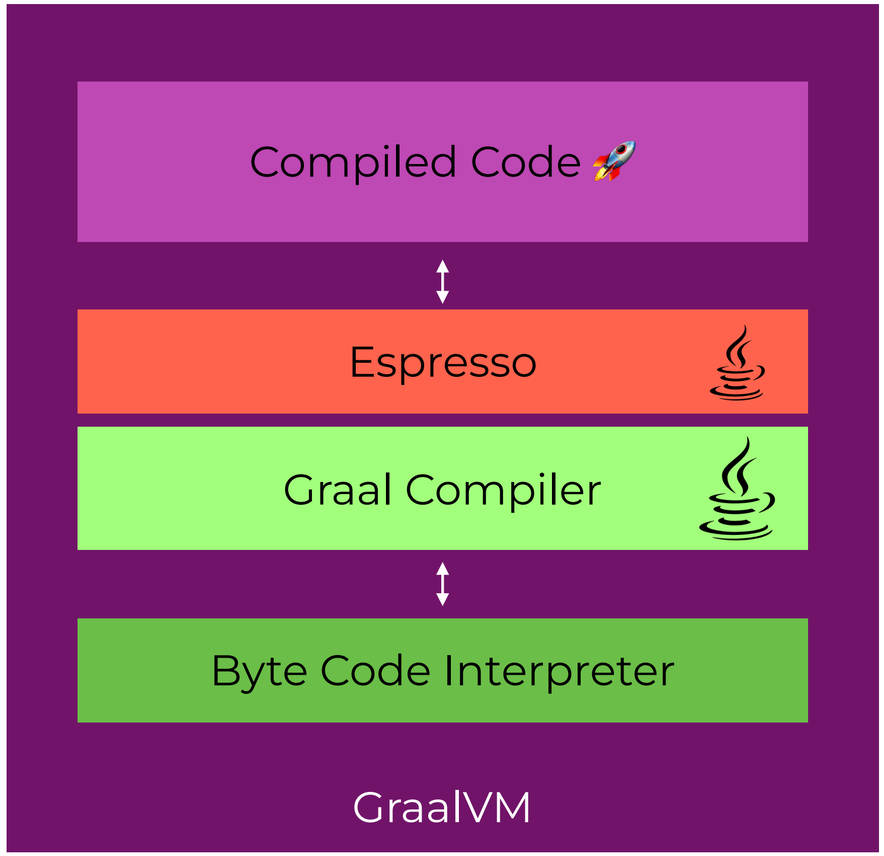

Our Application
8

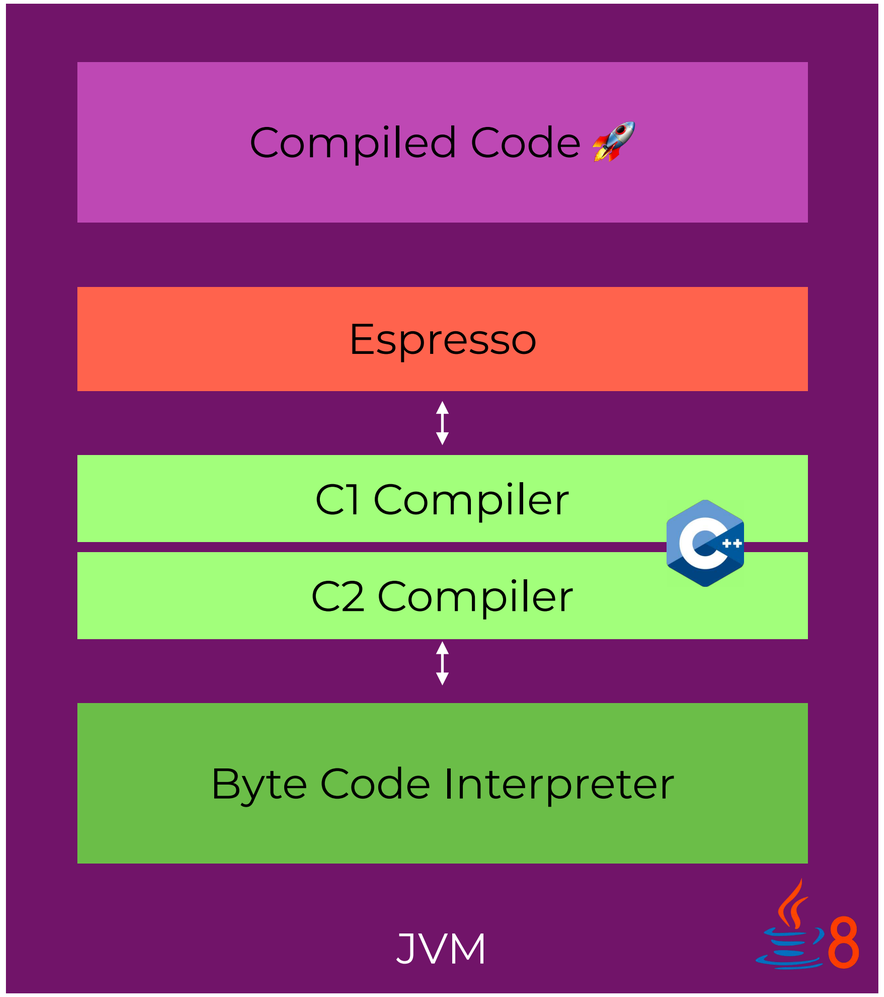

Java
JS
Espresso
Java
Truffle
GraalVM JIT Compiler
GraalVM Runtime (SubstratVM)

espresso



GraalVM in Just-In-Time (JIT) Compilation Mode


Compiled Code 🚀
Espresso
Graal Compiler
Byte Code Interpreter
GraalVM


Compiled Code 🚀
Espresso
C1 Compiler
C2 Compiler
Byte Code Interpreter
JVM
8

Espresso

Espresso

What performance can I expect from executing Java on Truffle?

Performance is currently 2-3x slower than HotSpot. It does not match the speed offered by GraalVM yet for sure, but having created a fully working Java on Truffle runtime, the development team is now focusing on making it as performant as the GraalVM JIT.

Does Java running on Truffle run on HotSpot too?

Like other languages implemented with the Truffle framework, it can run both as a native executable or on top of HotSpot. Running on top of HotSpot is currently only possible on Linux. We plan to extend this capability to macOS and Windows platforms also.

```
Exception in thread "main" Espresso cannot evaluate Java sources directly, only a few special commands are
supported: <Bindings> and <ProcessReferences>
Use the "java" language bindings to load guest Java classes e.g.
context.getBindings("java").getMember("java.lang.Integer")
```

Native JShell and Espresso: Mixing AOT and JIT for Java

This demo showcases the integration between GraalVM Native Image and Java-on-Truffle (Espresso).
It builds a native executable version of `jshell`, that executes the dynamically generated bytecodes on Espresso. This hybrid mode achieves instant startup, beating the vanilla `jshell` in both: time to the first interaction and time to evaluate a simple expression.

JShell is a Java read-eval-print loop tool first introduced in Java 9, this demo also allows running `jshell` on Java 8.
For further discussions and questions please join our `#espresso` channel on the GraalVM Slack Community.

Prerequisites

- GraalVM for Java 11, 17 or higher
- Native Image support
- Java-on-Truffle (Espresso) support

Preparation

1. Download and install the latest GraalVM JDK with Native Image and Java on Truffle (Espresso) components using the GraalVM JDK Downloader.

   ```
   bash <(curl -sL https://get.graalvm.org/jdk) -c 'espresso'
   ```

   The download script will provide the `JAVA_HOME`, `PATH`, and `GRAALVM_HOME` environment variables (follow the command line prompt).

2. Download or clone GraalVM demos repository and navigate into the `espresso-jshell` directory:

Espresso

GraalVM in Just-In-Time (JIT) Compilation Mode

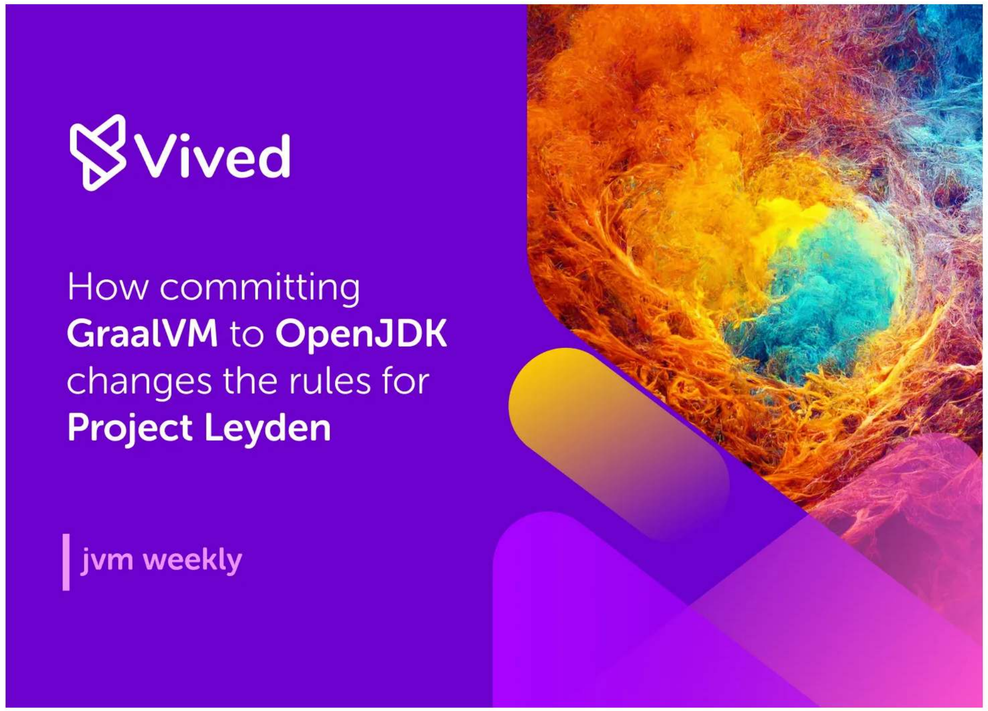
GraalVM in Just-In-Time (JIT) Compilation Mode

JVM Weekly (by vived.io)

How contributing GraalVM to OpenJDK changes the rules for Project Leyden - JVM Weekly #19

Instead of the standard division, I decided to prepare one coherent article, in which I will try to guide you through everything that is happening in Project Leyden and GraalVM.

Artur Skowronski
21 hr ago



Thank you 🙏

@ArturSkowronski