Graal & Truffle

How to write fast interpreters on the JVM

download: html / pdf

Manuel Leduc (https://mleduc.xyz/)

Domain Specific Language

> A domain-specific language (DSL) is a computer language specialized to a particular application domain

Examples

- awk
- pom
- CSS
- ...

To Read

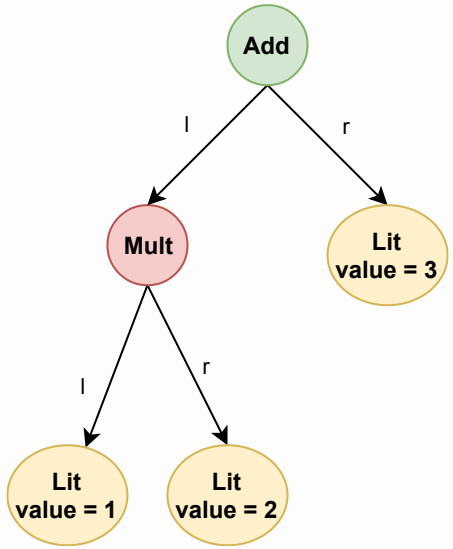
Martin fowler https://martinfowler.com/dsl.html

Domain Specific Language

Program

(1x2)+3

Abstract Syntax tree



Semantics

[Mult] = [l] * [r]

[Add] = [l] + [r]

[Lit] = value

Result = 6

Interpreter or Compiler?

Compiler

- **+** Fast
- **-** Hard to implement
- **-** Hard to reuse (compose, extends...)

Interpreter

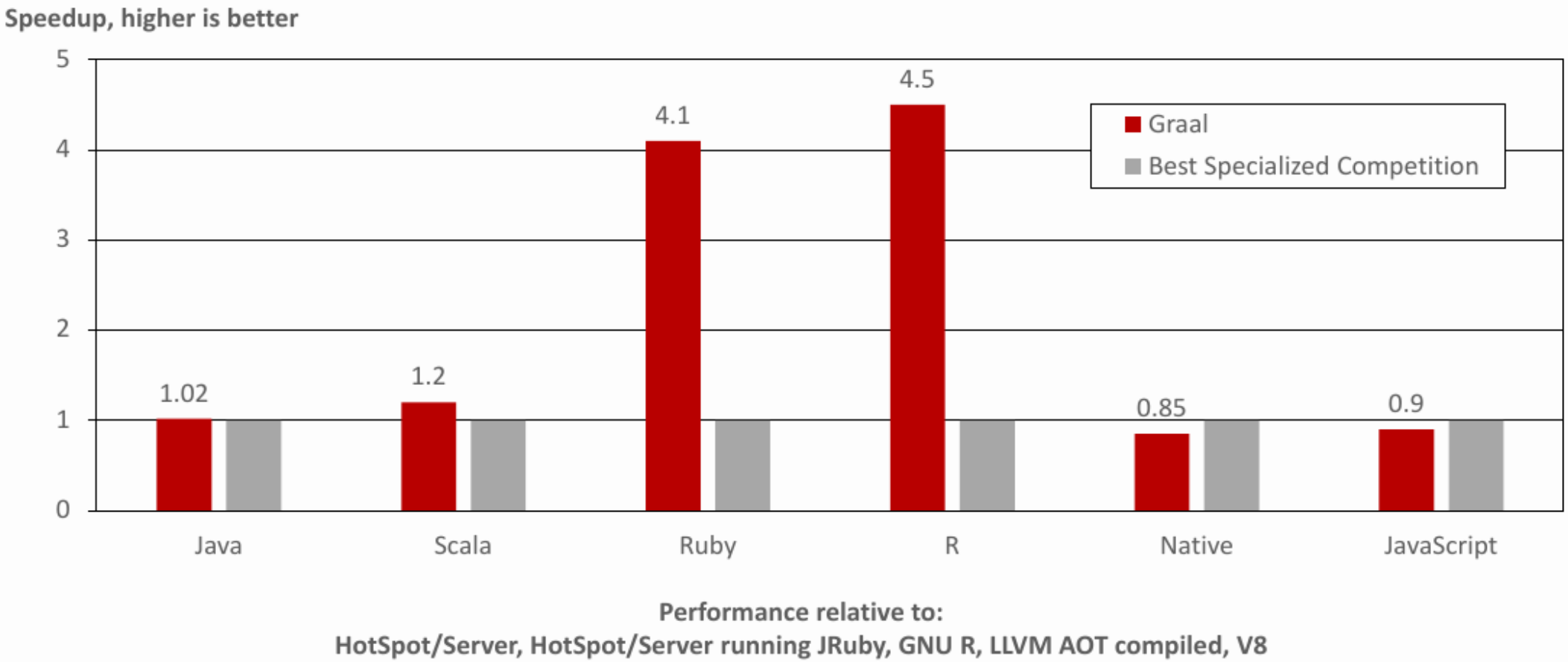
- **-** Slow(er)
- **+** Easier to implement
- **+** Easier to reuse (compose, extends...)

Objective

Obtaining interpreters with acceptable performances on the first try.

Graal

Performance



Just-In-Time Compiler and the JVM

Just-In-Time (JIT) Compiler

> Instead of interpreting bytecode instruction one by one, parts of the bycecode are compiled at runtime before being executed

JDK9 and JVMCI

JVMCI = Java-Level JVM Compiler Interface

C++ is replaced by Java

- Better **Modularity**
- Avoid **JVM (Re)-Compilation**

JVMCI internals

It *simply* takes a bunch of bytecode operations and yield equivalent machine level code.

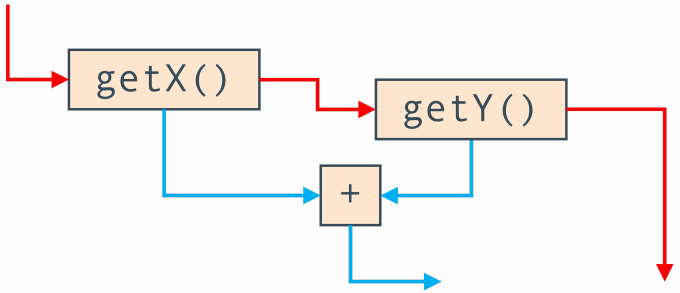
```
interface JVMCICompiler {
    byte[] compileMethod(byte[] bytecode);
}
```

Graal

> Graal is a JIT implementation build on top of the `JVMCICompiler` interface.

Sea of Node graph

Graal is based on a "sea of nodes" connected by Data Flow (blue) and Control Flow (red) annotations.

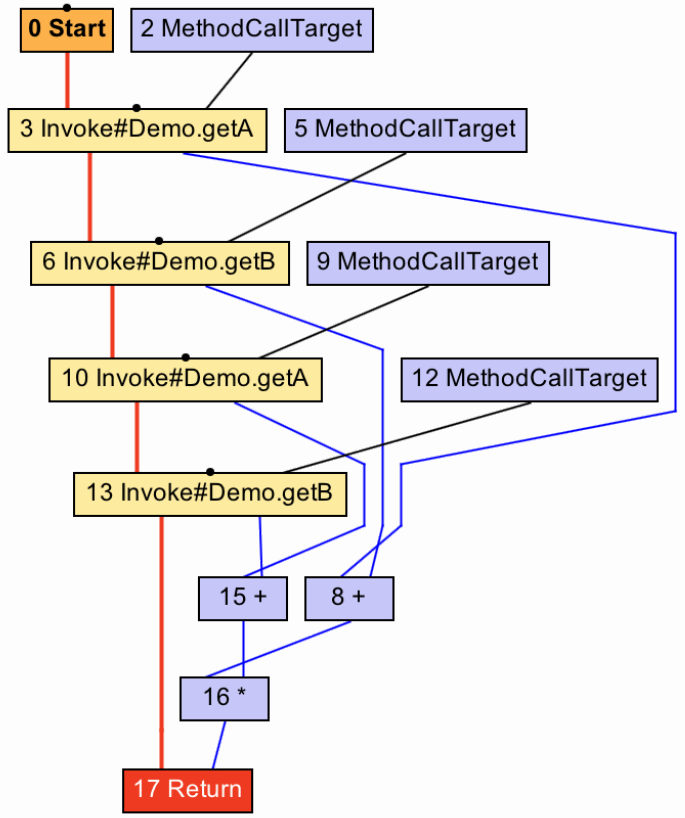


Partial interpretation

This graph is build by partially interpreting the bytecode.

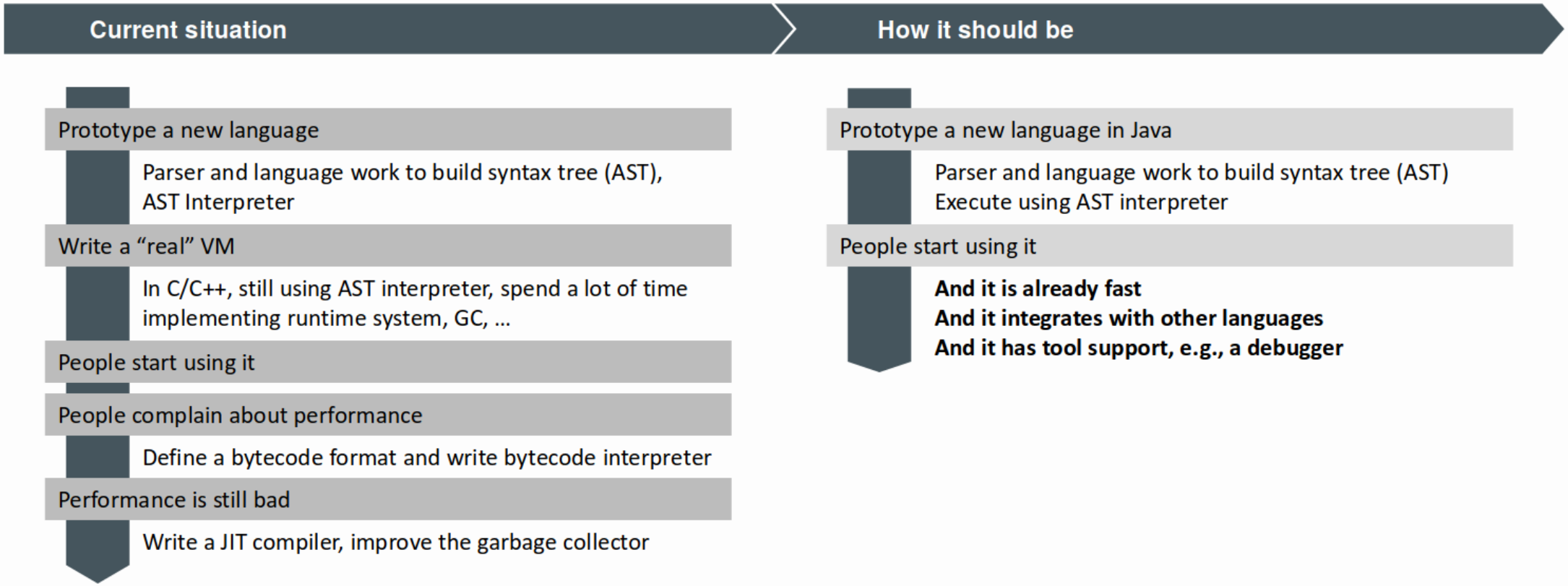
Graph Properties

- Annotated with runtime information
- Garbage collection analysis
- High-Level static optimizations
  - Method inlining
  - Constant folding
  - Arithmetic optimizations
- Speculative optimizations
- Can be visualy inspected



Truffle

Truffle in a picture



Truffle ideas

- High level representation of the language AST
- Annotated AST to assist **Graal** speculative optimizations.

Language implementation with Truffle

Simple concepts

- nodes
- methods specialization
- "classical" interpreters
- language metadata
- stack frames

```
@NodeInfo(shortName = "+")
public abstract class ELTAddNode extends ELTBinaryNode {

    @Specialization
    protected int add(int left, int right) {
        return left + right;
    }

    @Specialization
    protected float add(float left, float right) {
        return left + right;
    }

    @Specialization
    protected String add(String left, String right) {
        return left + right;
    }

    @Specialization
    protected int add(Object left, int right) {
        return ((Integer) left) + right;
    }
}
```

Polyglot

Example

```
#include <polyglot.h>

int main() {
    void *array = polyglot_eval("js", "[1,2,42,4]");
    int element = polyglot_as_i32(polyglot_get_array_element(array, 2));
    printf("%d\n", element);
    return element;
}
```

To Read

http://www.graalvm.org/docs/reference-manual/polyglot/

Scientific questions

- Can we reuse Truffle to define interpreters using ALE/K3/Gemoc... solutions?
- What new can we do if everything run *fast* on the JVM?
- What is the consequence of Polyglot for the DSL intercompatibility?

Resources

- **Really good blog post:** http://chrisseaton.com/truffleruby/jokerconf17/
- **Another blog post:** https://zeroturnaround.com/rebellabs/graal-and-truffle-for-polyglot-languages-on-jvm/
- **Comprehensive slides:** http://lafo.ssw.uni-linz.ac.at/papers/2017_PLDI_GraalTutorial.pdf
- **Publications:** https://github.com/oracle/graal/blob/master/docs/Publications.md