GraalVM™

Truffle Metacompilation in Action

Christian Humer
Oracle Labs Zürich

About Me

Christian Humer

Truffle Framework Lead

Industry Researcher / VM engineer
Oracle Labs Zurich

Started @JKU Linz
Working on GraalVM for ~12 years.

What is Oracle GraalVM?

Drop-in replacement for Oracle Java

- Run your Java application faster
- More just-in-time (JIT) compiler optimizations

Ahead-of-time (AOT) compilation for Java

- Create standalone binaries with low footprint

High-performance Polyglot VM ...

- Implement your own language or DSL

Based on more than a decade of research from Oracle Labs

What to expect today?

Theory!

Interpretation vs Compilation

Manual vs Metacompilation

Futamura Projections

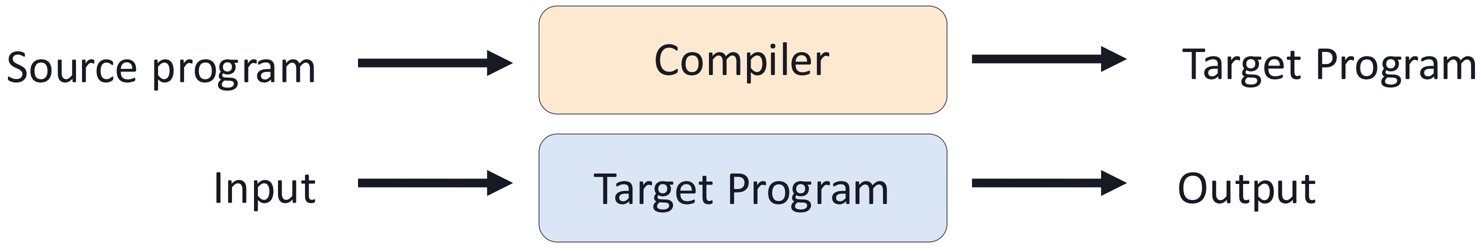
Metacompilation with Truffle

Practice!

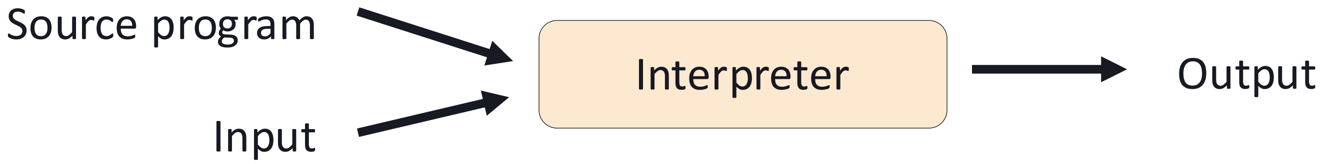
Build our own language called TinyLang

```
; (1 + 2) + 3
(add (add 1 2) 3)
```

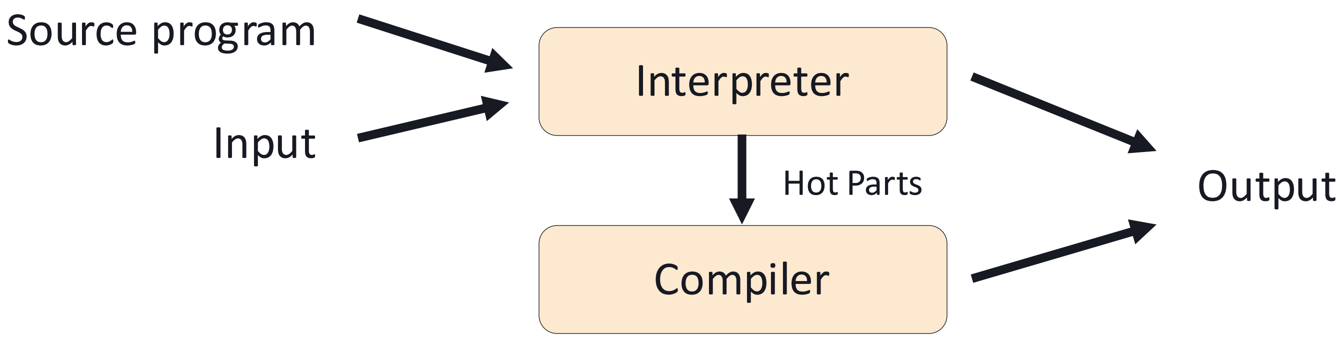
Pure Compilation



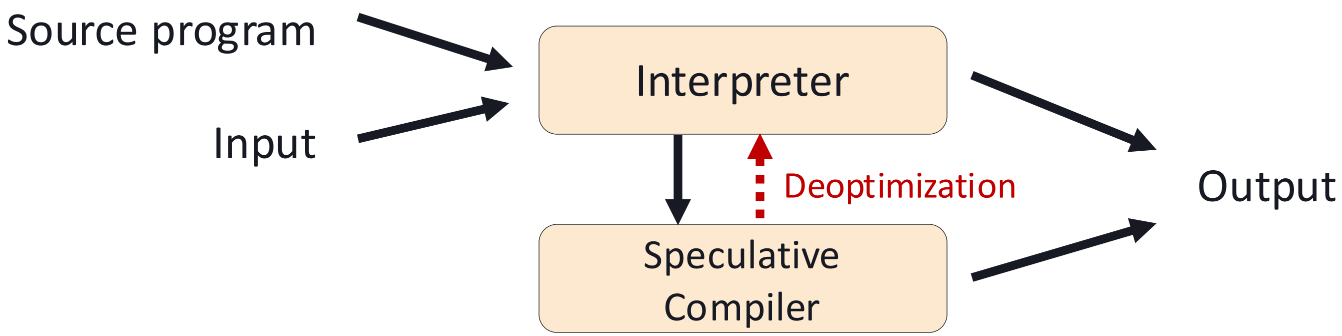
Pure Interpretation



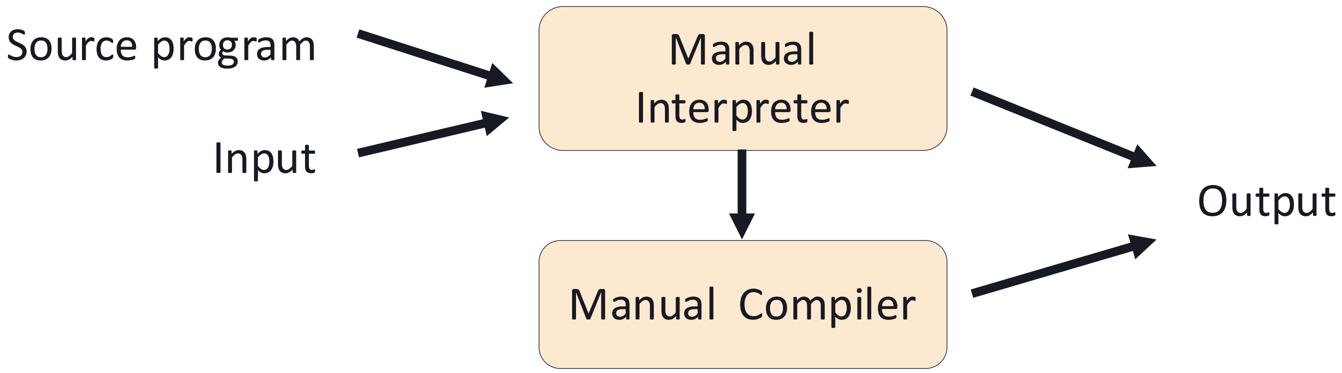
Just in Time Compilation



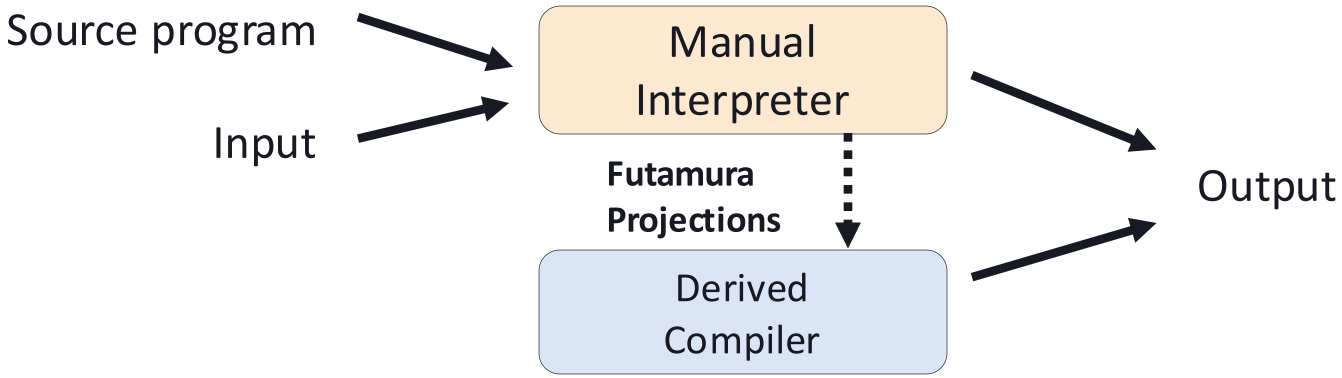
Dynamic Compilation



Manual JIT Compilation



Meta JIT Compilation



Futamura Projections

Partial Evaluation of Computation Process— An Approach to a Compiler-Compiler

YOSHIHIKO FUTAMURA
Central Research Laboratory, Hitachi, Ltd., Kokubunji, Tokyo, Japan 185

Systems.Computers.Controls, Volume 2, Number 5, 1971

Abstract. This paper reports the relationship between formal description of semantics (i.e., interpreter) of a programming language and an actual compiler. The paper also describes a method to automatically generate an actual compiler from a formal description which is, in some sense, the partial evaluation of a computation process. The compiler-compiler inspired by this method differs from conventional ones in that the compiler-compiler based on our method can describe an evaluation procedure (interpreter) in defining the semantics of a programming language, while the conventional one describes a translation process.

Partial Evaluation

Function
Static Parameters
→ Partial Evaluator → Specialized Function

1st Futamura Projection

Interpreter Function
Source program
(e.g. AST, Bytecodes)
→ Partial Evaluator → Program Executable

Focus of today!

Truffle Framework

- Meta Compilation Using Futamura Projections
- Dynamic Speculation and Automatic Deoptimization
- Language Composition with Interop
- Tool Composition with Instrumentation
- Polyglot Embedding and Sandboxing

Focus of today!

Truffle Interpreter Example

```
abstract class Expression extends Node {
    abstract int execute(int[] arguments);
}

class Add extends Expression {
    @Child Expression left;
    @Child Expression right;

    Add(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    int execute(int[] args) {
        return left.execute(args) + right.execute(args);
    }
}
```

```
class Arg extends Expression {
    final int index;

    Arg(int index) { this.index = index; }

    int execute(int[] args) {
        return args[index];
    }
}
```

```
int interpret(Expression expression, int[] args) {
  return expression.execute(args);
}
```

```
// Sample program (arg[0] + arg[1]) + arg[2]
sample = new Add(new Add(new Arg(0), new Arg(1)), new Arg(2));
```

Truffle Interpreter Compilation

```
// Sample program (arg[0] + arg[1]) + arg[2]
sample = new Add(new Add(new Arg(0), new Arg(1)), new Arg(2));
```

```
int interpret(Expression expression, int[] args) {
   return expression.execute(args);
}
```

```
partiallyEvaluate(interpret, sample)
```

```
int interpretSample(int[] args) {
    return sample.execute(args);
}
```

1st Futamura Projection

Truffle Interpreter Compilation

```
// Sample program (arg[0] + arg[1]) + arg[2]
sample = new Add(new Add(new Arg(0), new Arg(1)), new Arg(2));
```

```
int interpretSample(int[] args) {
    return sample.execute(args);
}
```

```
int interpretSample(int[] args) {
    return sample.left.execute(args)
         + sample.right.execute(args);
}
```

```
int interpretSample(int[] args) {
    return sample.left.left.execute(args)
         + sample.left.right.execute(args)
         + args[sample.right.index];
}
```

```
int interpretSample(int[] args) {
    return args[sample.left.left.index]
         + args[sample.left.right.index]
         + args[sample.right.index];
}
```

```
int interpretSample(int[] args) {
    return args[0]
         + args[1]
         + args[2];
}
```

Initiate Partial Evaluation

```
class Function extends RootNode {
    @Child Expression body;
    ...
    @Override
    Object execute(VirtualFrame frame) {
        return body.execute((int[])frame.getArguments()[0])
    }
}

public static void main(String[] args) {
    Function sample = new Function(new Add(new Add(new Arg(0), new Arg(1)), new Arg(2)));
    CallTarget target = sample.getCallTarget();
    for (int i = 0; i < 1000; i++) {
      target.call(new int[] { 10, 11, 21 });
    }
    System.out.println(“done“);
}
```

Demo Partial Evaluation

(https://github.com/chumer/pedemo)

AST vs. Bytecode interpreters

AST interpreters

- struggle with memory footprint (must instantiate an entire tree of nodes)
- are difficult to optimize in the interpreter (many polymorphic execute calls)

Bytecode interpreters

- enjoy the same peak performance
- can densely encode programs (just bytecode)
- can implement instructions uniformly (→ template JIT)
- simplify complex control flow (including continuations)
- are difficult to write ☹

Goal: *generate bytecode interpreters automatically*

Bytecode DSL

```
def triple(x):
  return x * 3
```

```
b.beginRoot(...);
 b.beginReturn();
  b.beginMultiply();
   b.emitLoadLocal(...);
   b.emitLoadConstant(3);
  b.endMultiply();
 b.endReturn();
b.endRoot();
```

```
LOAD_LOCAL, ...
LOAD_CONST, ...
MULTIPLY, ...
RETURN
```

Use any parser you like

AST-like builder with custom node-like operations

Automatic bytecode format and optimization

Demo Bytecode Interpreters

(https://github.com/chumer/pedemo)

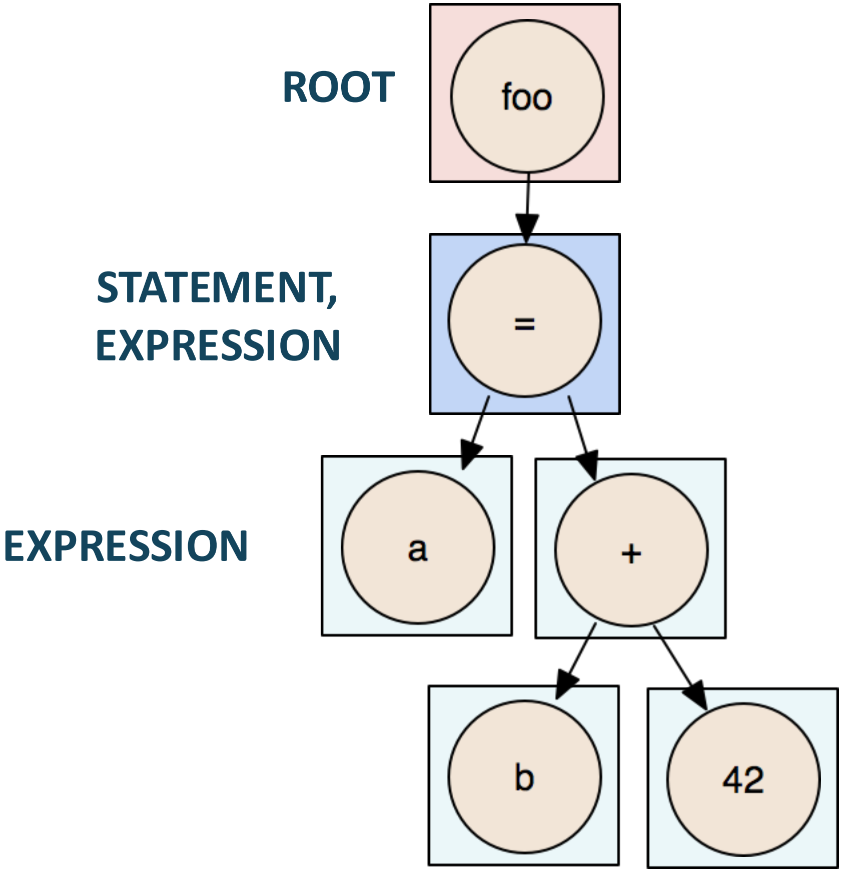
Demo TinyLang Part 1

(https://github.com/chumer/tinylang/)

```
; (1 + 2) + 3
(add (add 1 2) 3)
```

Tag Instrumentation

```
function foo(a, b) {
    a = b + 42
}
```



How do we tag TinyLanguage?

```
; (1 + 2) + 3
(add (add 1 2) 3)
```

Tag Instrumentation for Bytecode Interpreters

```
// Source: function() {return 42;}
parse((b) -> {
        b.beginRoot(slLanguage);

        b.beginReturn();
        b.beginTag(StatementTag.class);
        b.beginTag(ExpressionTag.class);
        b.emitLoadConstant(42);
        b.endTag(ExpressionTag.class);
        b.endTag(StatementTag.class);
        b.endReturn();

        b.endRoot();
});
```

Without tags

```
UncachedBytecodeNode()
 instructions(2) =
   [00] 0b load.constant    constant(Integer 42)
   [02] 06 return
 exceptionHandlers(0) = Empty
 sourceInformation(0) = Empty
 tagTree = Not Available
]
```

Transition on-stack + replay of events

With all tags

```
UncachedBytecodeNode()
 instructions(10) =
   [00] 1f tag.enter     tag(00-12, ROOT_BODY, ROOT)
   [02] 1f tag.enter     tag(02-0b, STATEMENT)
   [04] 1f tag.enter     tag(04-08, EXPRESSION)
   [06] 0b load.constant  constant(Integer 42)
   [08] 20 tag.leave     tag(04-08, EXPRESSION)
   [0b] 20 tag.leave     tag(02-0b, STATEMENT)
   [0e] 20 tag.leave     tag(00-12, ROOT_BODY, ROOT)
   [11] 06 return
   [12] 20 tag.leave     tag(00-12, ROOT_BODY, ROOT)
   [15] 06 return
 exceptionHandlers(3) =
   [04 : 0b] -> tag.exceptional tag(04-08, EXPRESSION)
   [02 : 0e] -> tag.exceptional tag(02-0b, STATEMENT)
   [00 : 15] -> tag.exceptional tag(00-12, ROOT_BODY, ROOT)
 sourceInformation(0) = Empty
 tagTree(3) =
   Node[00-12, ROOT_BODY, ROOT]
     Node[02-0b, STATEMENT]
       Node[04-08, EXPRESSION]
```

Features:

- On-demand materialization of individual sets of tags by reparsing.
- On stack replacement for tags
- No runtime overhead if not used. No additional read in the BC loop
- Full instrumentation support (unwind, restart, input values)

Tooling

- **Chrome Inspector** – Debug and profile languages in Chrome Inspector

Focus of today!

- **CPUSampler** - Sample time spent in optimized and interpreted code.
- **Coverage** - Produce coverage results across languages (e.g. lcov, raw)
- **Visual Studio Code Integration** – For polyglot language developement
- **Language Server Protocol** (experimental) – Dynamic Autocomplete
- **GraalVM Insight** (experimental) – Tracing Using Scripts

https://www.graalvm.org/docs/tools/

Demo TinyLang Part 2

(https://github.com/chumer/tinylang/)

Graal Languages based on Truffle

Oracle Developed

JS

python

Ruby

WA

Scala

Groovy

Java

Espresso

Sulong

RegEx

Externally Developed

TruffleSqueak

PKL

Enso

Yona

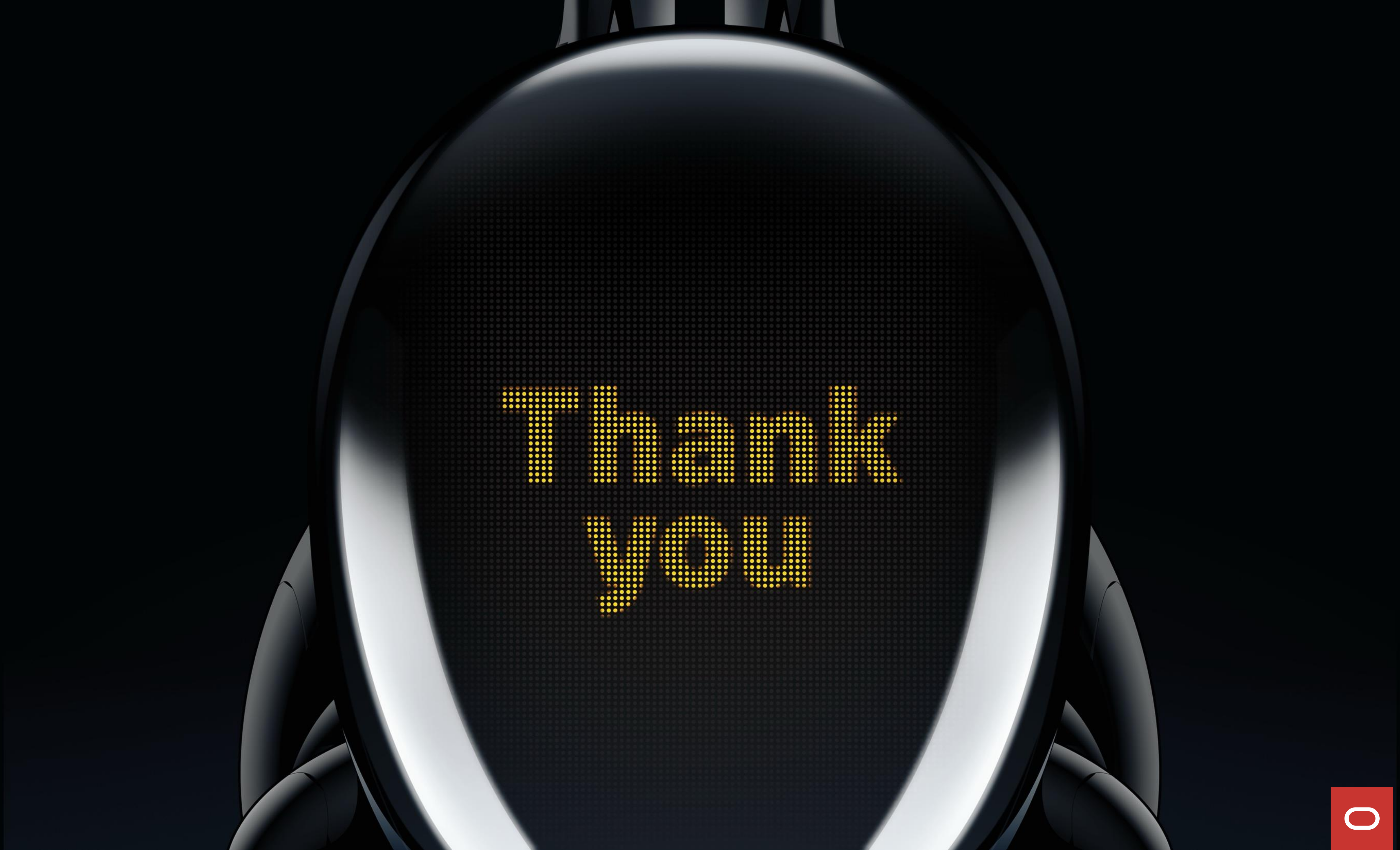
NVIDIA CUDA

SOM SimpleObjectMachine

Goldman Sachs

Slang

All languages: https://github.com/oracle/graal/blob/master/truffle/docs/Languages.md


Thank
you