ORACLE®

One VM to Rule Them All

Christian Wimmer

VM Research Group, Oracle Labs

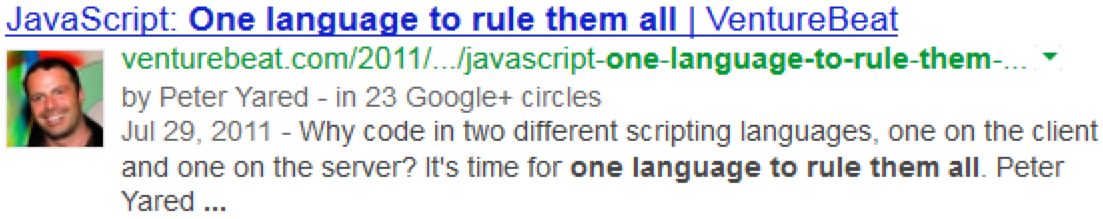
ORACLE

Safe Harbor Statement

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

One Language to Rule Them All?

Let’s ask a search engine…

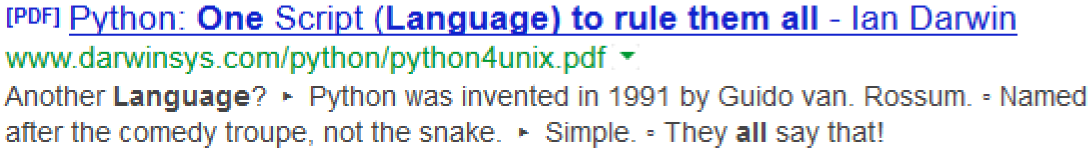


JavaScript: **One language to rule them all** | VentureBeat

venturebeat.com/2011/.../javascript-**one-language-to-rule-them**-...

by Peter Yared - in 23 Google+ circles

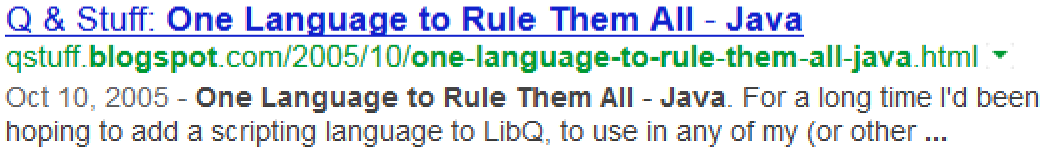
Jul 29, 2011 - Why code in two different scripting languages, one on the client and one on the server? It's time for **one language to rule them all**. Peter Yared **...**



[PDF] Python: **One** Script (**Language**) **to rule them all** - Ian Darwin

www.darwinsys.com/python/python4unix.pdf

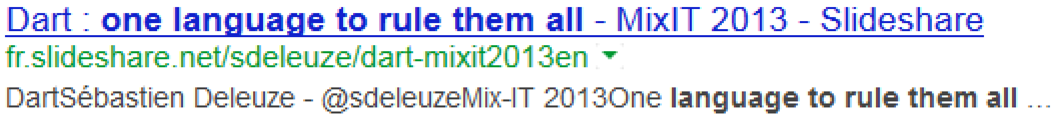
Another **Language**? ▸ Python was invented in 1991 by Guido van. Rossum. ▪ Named after the comedy troupe, not the snake. ▸ Simple. ▪ They **all** say that!



Q & Stuff: **One Language to Rule Them All - Java**

qstuff.**blogspot**.com/2005/10/**one-language-to-rule-them-all-java**.html

Oct 10, 2005 - **One Language to Rule Them All** - **Java**. For a long time I'd been hoping to add a scripting language to LibQ, to use in any of my (or other **...**



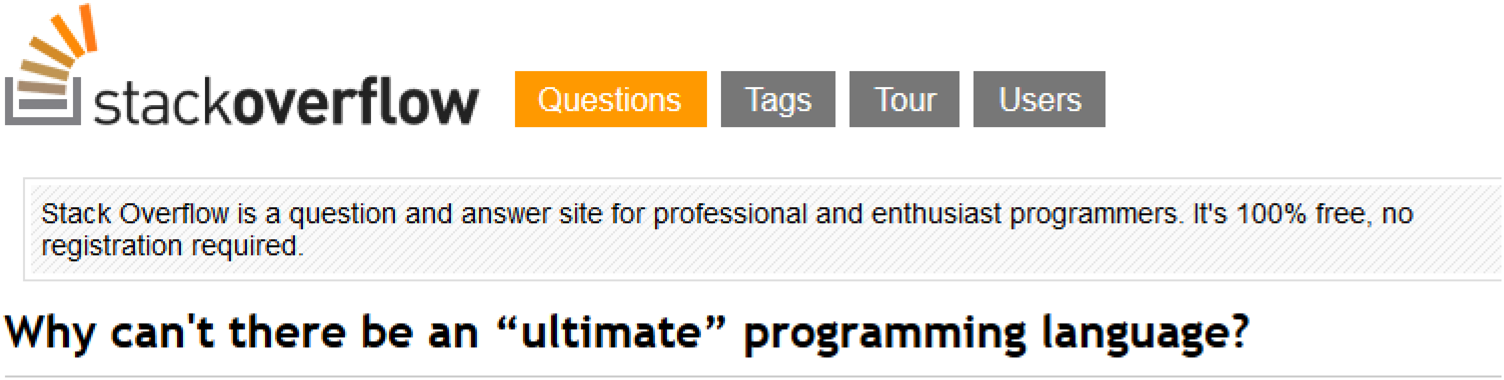
Dart : **one language to rule them all** - MixIT 2013 - Slideshare

fr.slideshare.net/sdeleuze/dart-mixit2013en

DartSébastien Deleuze - @sdeleuzeMix-IT 2013One **language to rule them all** ...

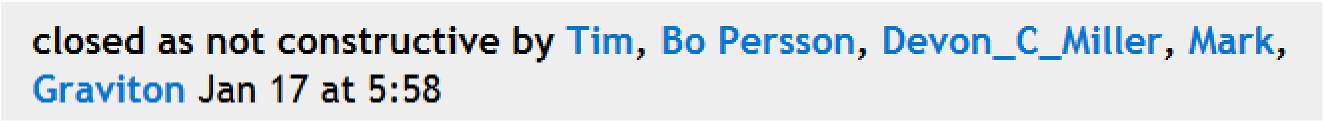
One Language to Rule Them All?

Let’s ask Stack Overflow…

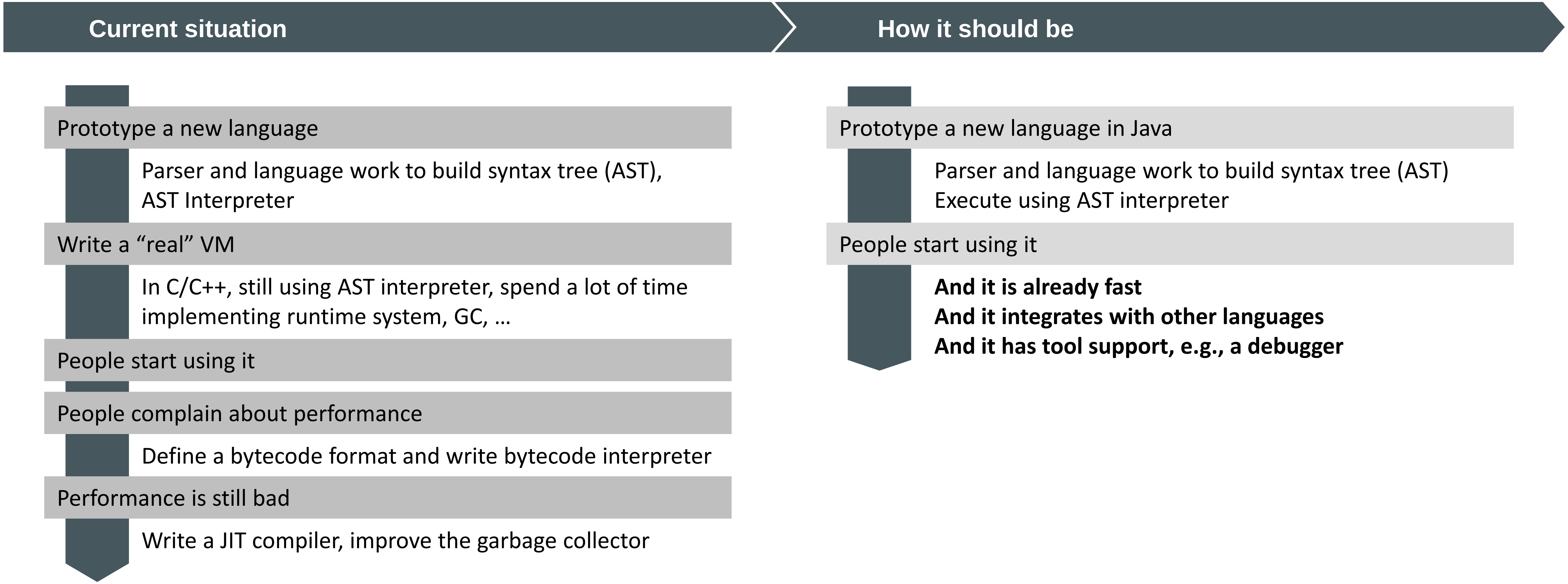


Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, no registration required.

Why can't there be an “ultimate” programming language?


closed as not constructive by Tim, Bo Persson, Devon_C_Miller, Mark, Graviton Jan 17 at 5:58

“Write Your Own Language”



Current situation

Prototype a new language

Parser and language work to build syntax tree (AST),
AST Interpreter

Write a “real” VM

In C/C++, still using AST interpreter, spend a lot of time implementing runtime system, GC, …

People start using it

People complain about performance

Define a bytecode format and write bytecode interpreter

Performance is still bad

Write a JIT compiler, improve the garbage collector

How it should be

Prototype a new language in Java

Parser and language work to build syntax tree (AST)
Execute using AST interpreter

People start using it

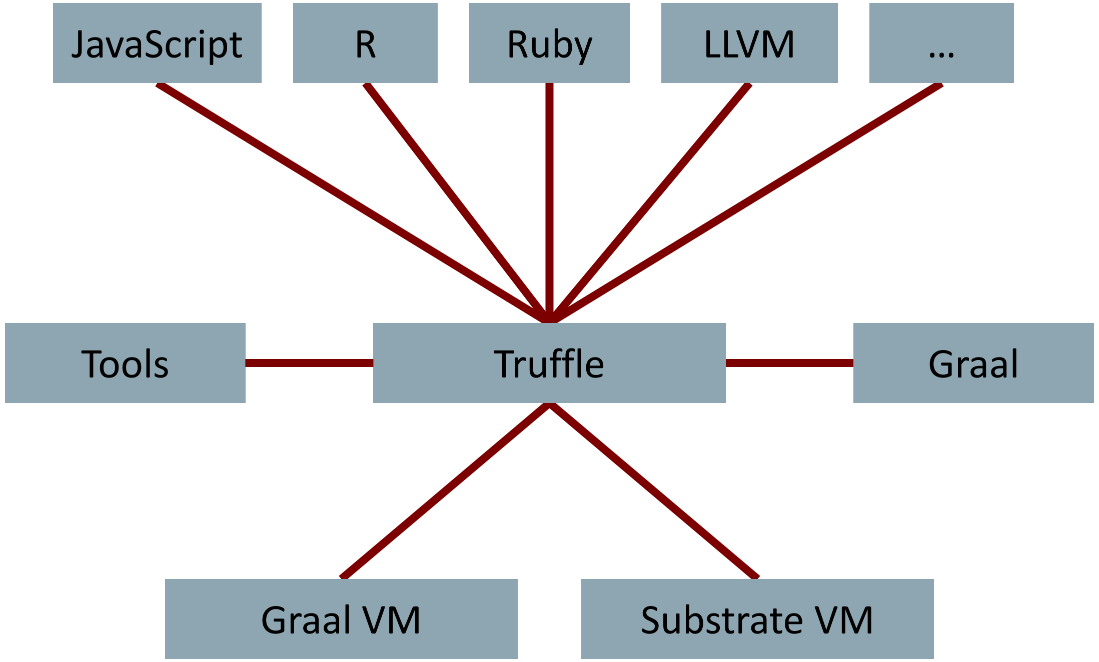
And it is already fast
And it integrates with other languages
And it has tool support, e.g., a debugger

ORACLE

Truffle System Structure

AST Interpreter for every language

Your language should be here!



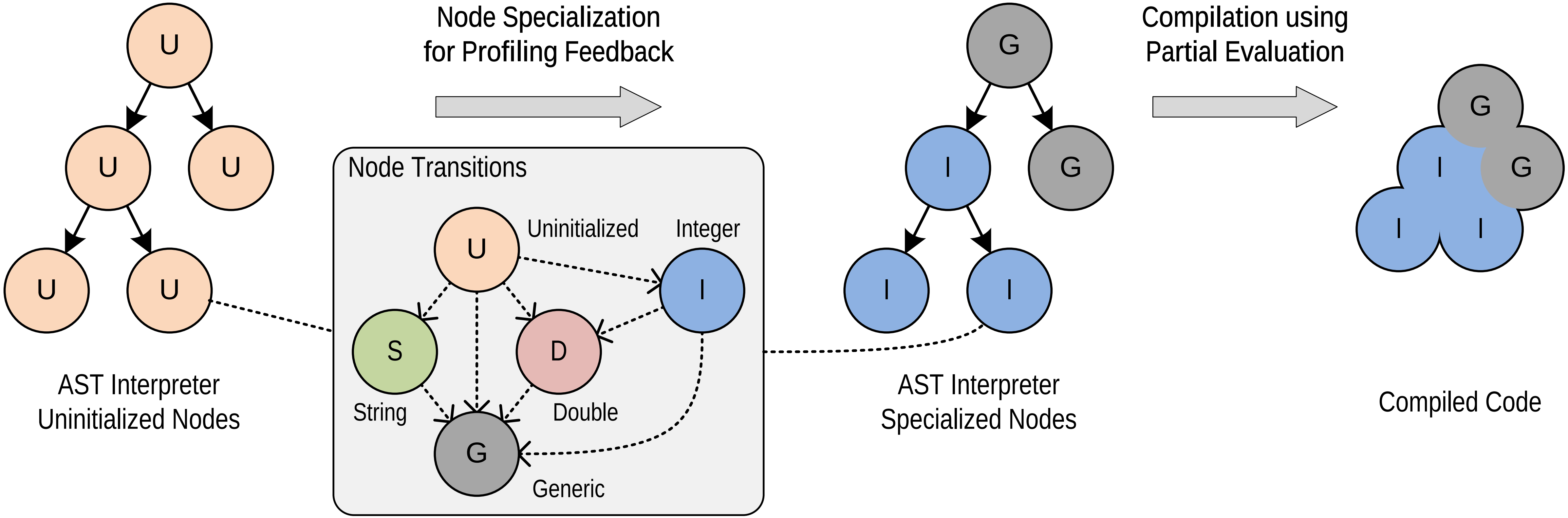
Common API separates language implementation, optimization system, and tools (debugger)

Language agnostic dynamic compiler

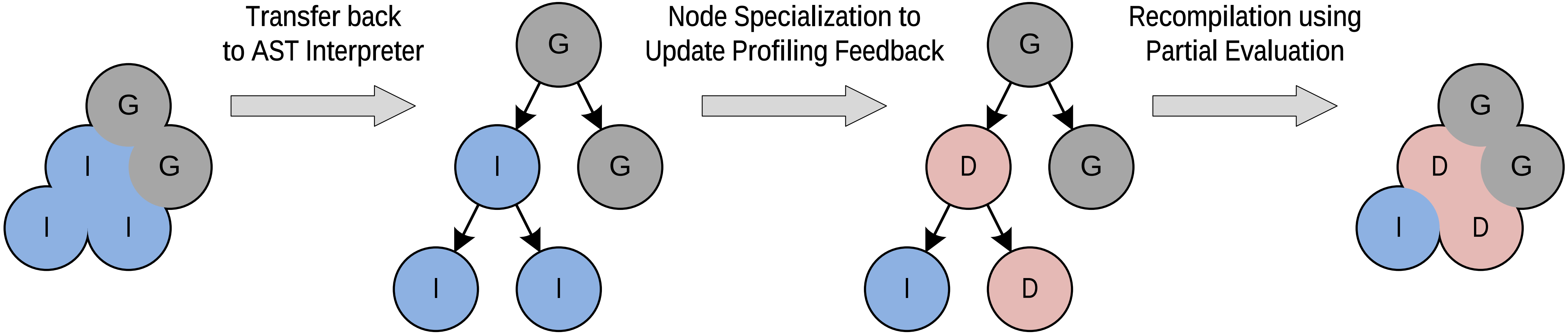
Integrate with Java applications

Low-footprint VM, also suitable for embedding

Speculate and Optimize …

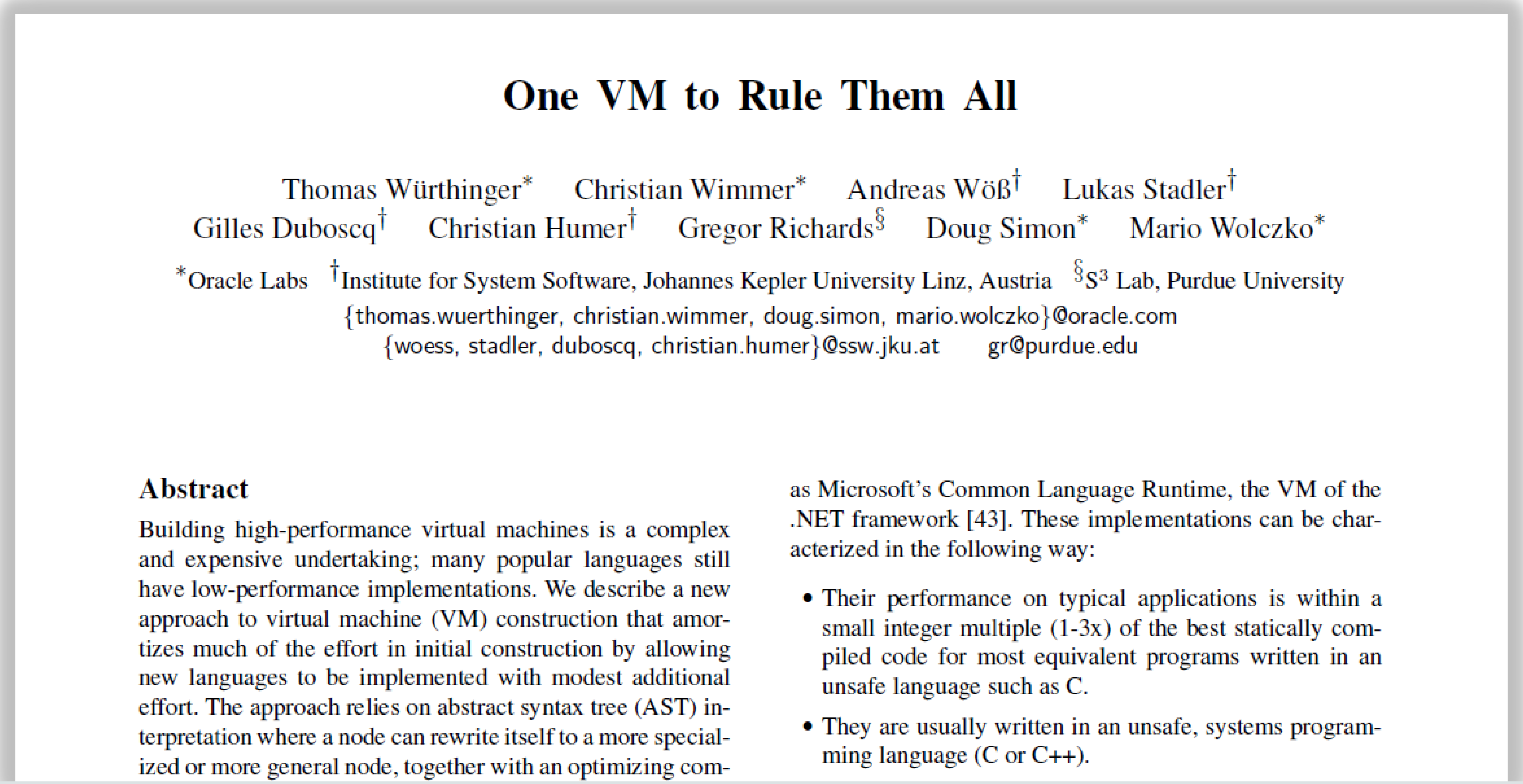


… and Transfer to Interpreter and Reoptimize!



More Details on Truffle Approach

https://wiki.openjdk.java.net/display/Graal/Publications+and+Presentations



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 [§]S^3 Lab, Purdue University
{thomas.wuerthinger, christian.wimmer, doug.simon, mario.wolczko}@oracle.com
{woess, stadler, duboscq, christian.humer}@ssw.jku.at gr@purdue.edu

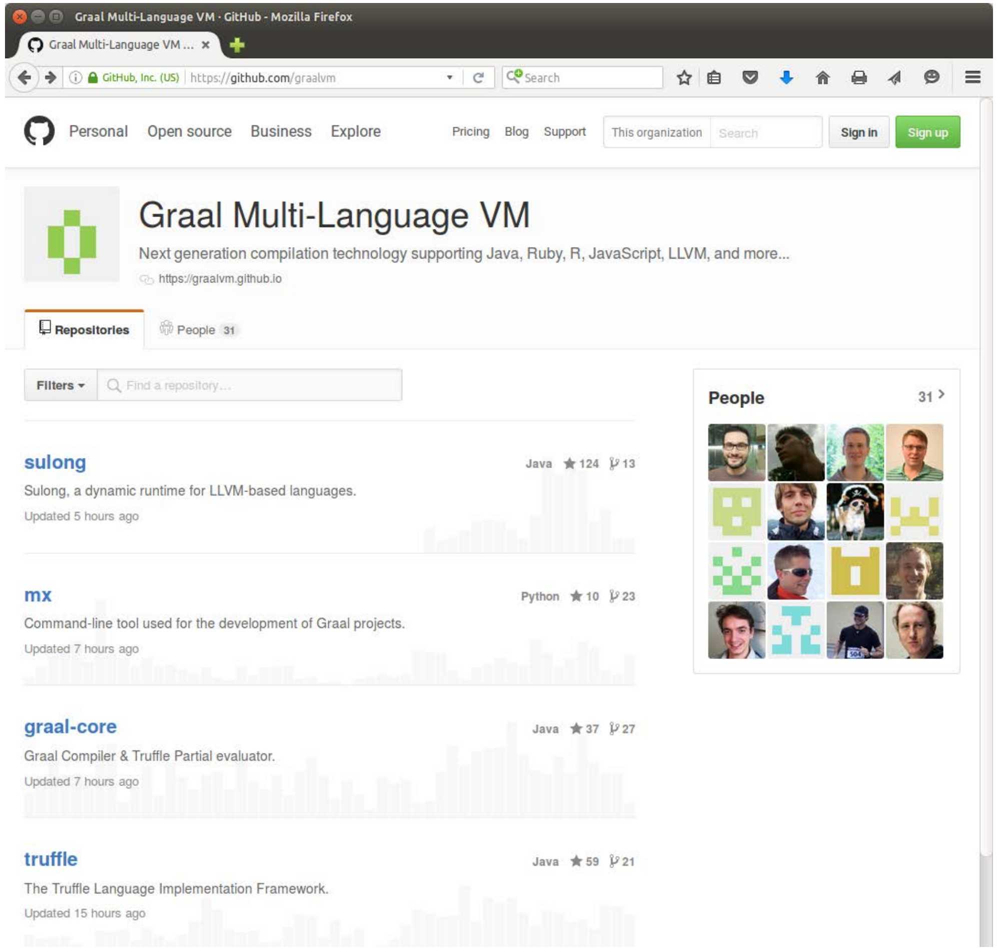
Abstract

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 com-

as Microsoft's Common Language Runtime, the VM of the .NET framework [43]. These implementations can be characterized in the following way:

- Their performance on typical applications is within a small integer 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++).

Open Source Code on GitHub



Graal Multi-Language VM

Next generation compilation technology supporting Java, Ruby, R, JavaScript, LLVM, and more...

https://graalvm.github.io

sulong Java ★124 13

Sulong, a dynamic runtime for LLVM-based languages.

Updated 5 hours ago

mx Python ★10 23

Command-line tool used for the development of Graal projects.

Updated 7 hours ago

graal-core Java ★37 27

Graal Compiler & Truffle Partial evaluator.

Updated 7 hours ago

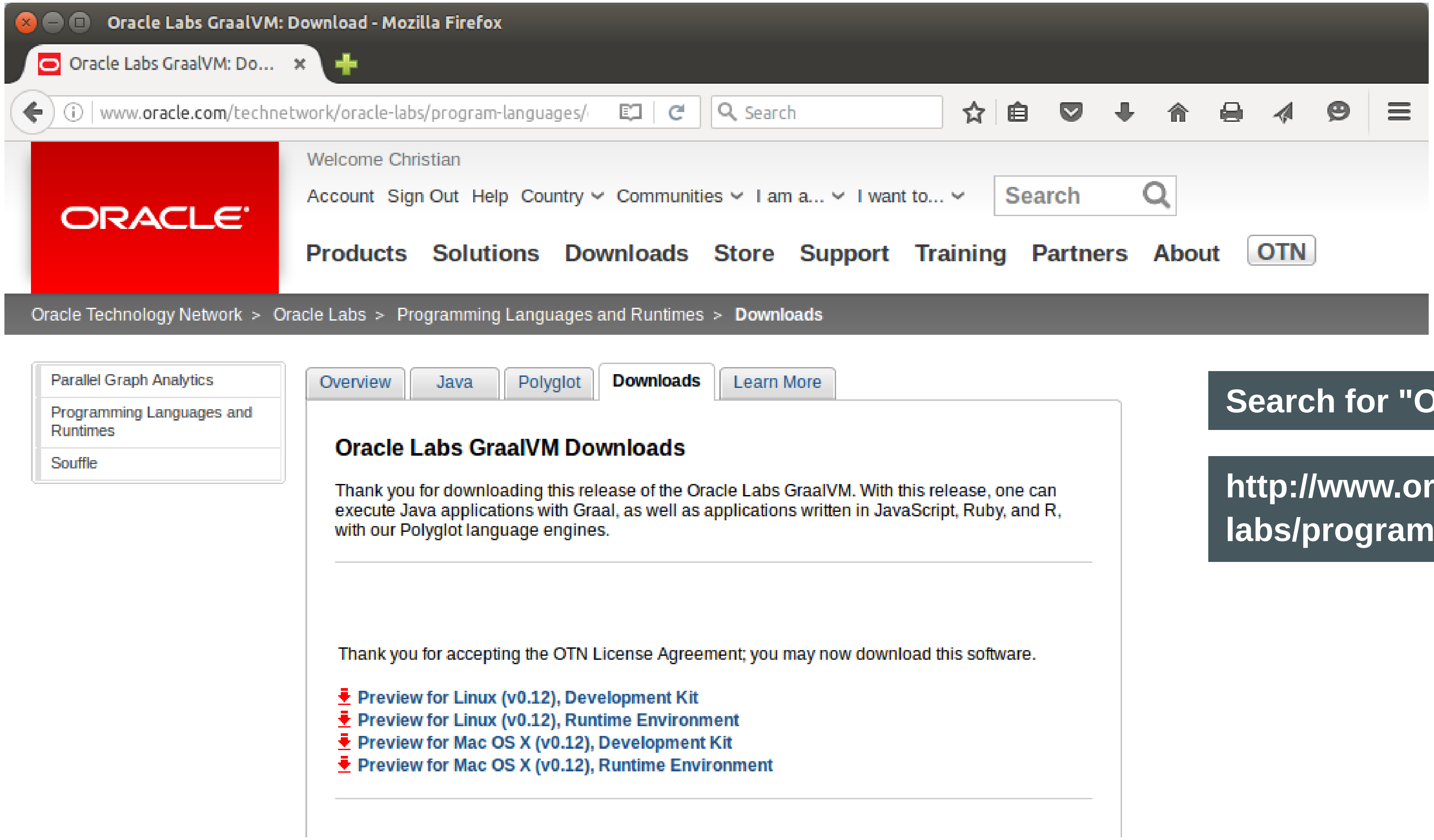
truffle Java ★59 21

The Truffle Language Implementation Framework.

Updated 15 hours ago

https://github.com/graalvm

Binary Snapshots on OTN



Oracle Labs GraalVM: Download - Mozilla Firefox
Oracle Labs GraalVM: Do...
www.oracle.com/technetwork/oracle-labs/program-languages/
Search
ORACLE
Welcome Christian
Account Sign Out Help Country Communities I am a... I want to...
Search
Products Solutions Downloads Store Support Training Partners About OTN
Oracle Technology Network > Oracle Labs > Programming Languages and Runtimes > Downloads
Parallel Graph Analytics
Programming Languages and Runtimes
Souffle
Overview Java Polyglot Downloads Learn More

Oracle Labs GraalVM Downloads

Thank you for downloading this release of the Oracle Labs GraalVM. With this release, one can execute Java applications with Graal, as well as applications written in JavaScript, Ruby, and R, with our Polyglot language engines.

Thank you for accepting the OTN License Agreement; you may now download this software.

- Preview for Linux (v0.12), Development Kit
- Preview for Linux (v0.12), Runtime Environment
- Preview for Mac OS X (v0.12), Development Kit
- Preview for Mac OS X (v0.12), Runtime Environment

Search for "OTN Graal"

http://www.oracle.com/technetwork/oracle-labs/program-languages/downloads/

Truffle Language Projects

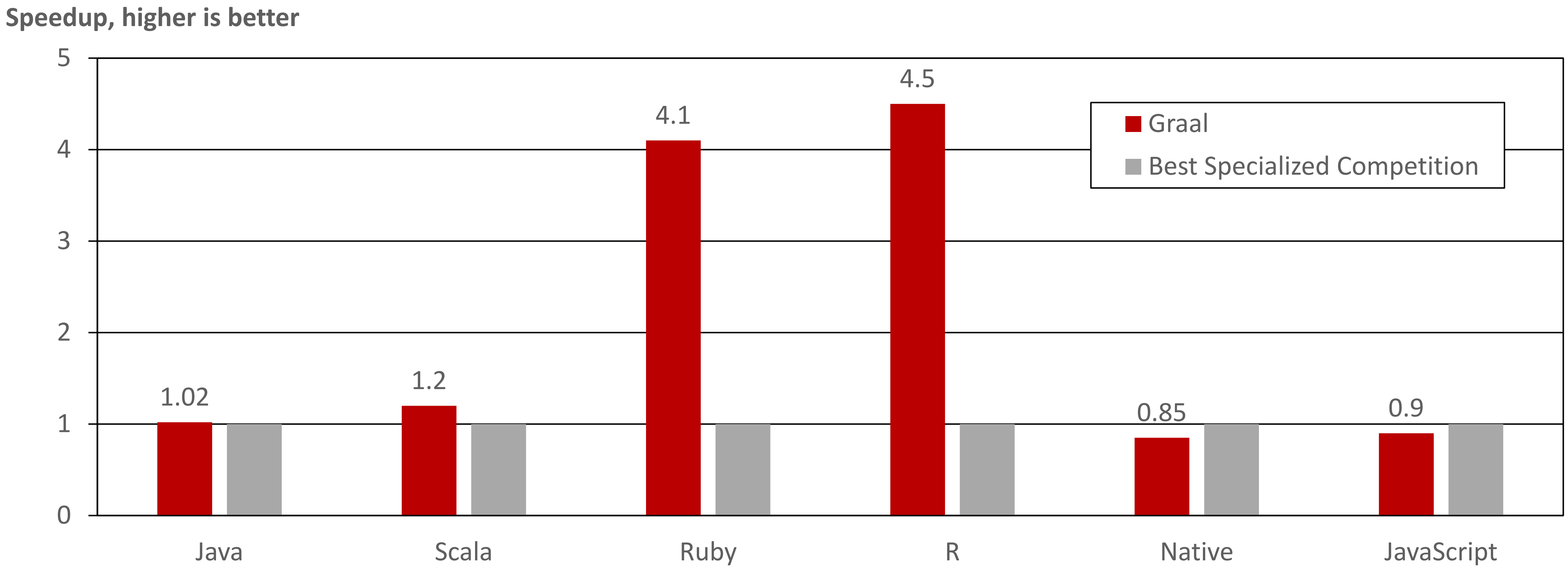
Some languages that we are aware of

- JavaScript: JKU Linz, Oracle Labs
  - http://www.oracle.com/technetwork/oracle-labs/program-languages/
- Ruby: Oracle Labs, included in JRuby
  - Open source: https://github.com/jruby/jruby
- R: JKU Linz, Purdue University, Oracle Labs
  - Open source: https://github.com/graalvm/fastr
- Sulong (LLVM Bitcode): JKU Linz, Oracle Labs
  - Open source: https://github.com/graalvm/sulong
- Python: UC Irvine
  - Open source: https://bitbucket.org/ssllab/zippy/
- SOM (Newspeak, Smalltalk): Stefan Marr
  - Open source: https://github.com/smarr/

Performance Disclaimers

- All Truffle numbers reflect a development snapshot
  - Subject to change at any time (hopefully improve)
  - You have to know a benchmark to understand why it is slow or fast
- We are not claiming to have complete language implementations
  - JavaScript: passes 100% of ECMAscript standard tests
    - Working on full compatibility with V8 for Node.JS
  - Ruby: passing 100% of RubySpec language tests
    - Passing around 90% of the core library tests
  - R: prototype, but already complete enough and fast for a few selected workloads
- Benchmarks that are not shown
  - may not run at all, or
  - may not run fast

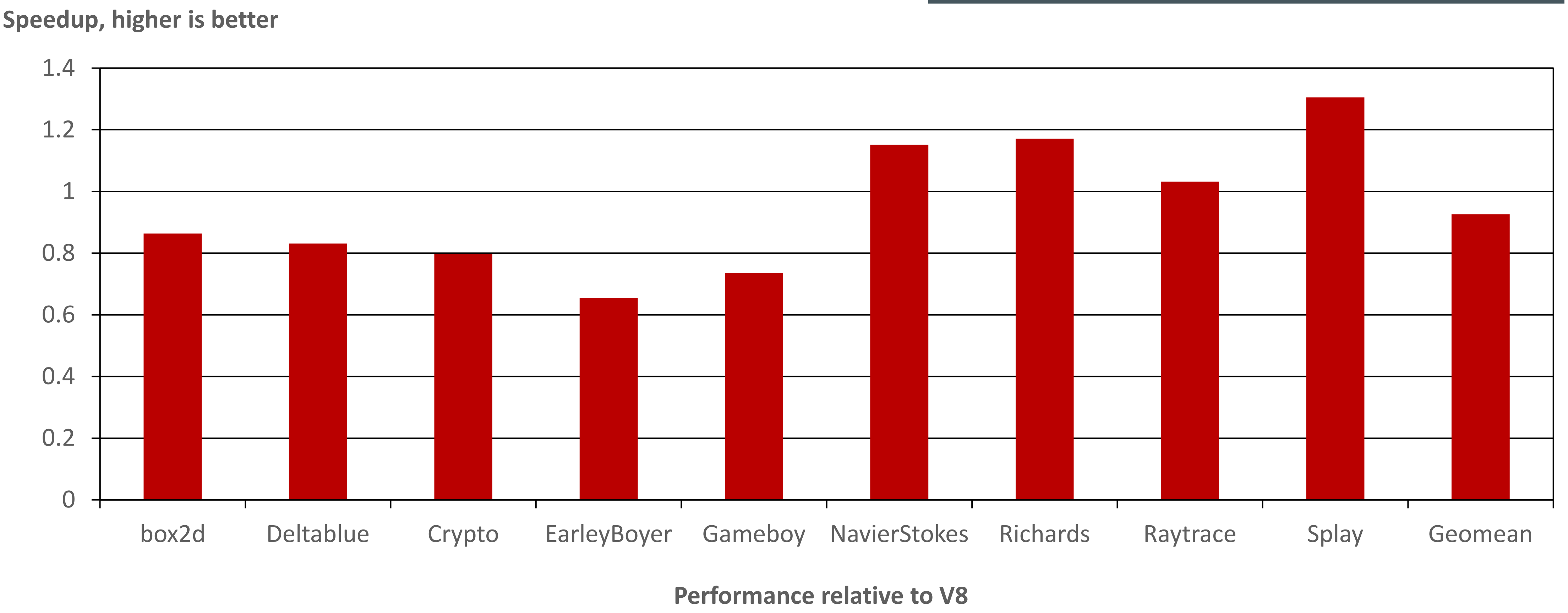
Performance: GraalVM Summary



Performance relative to:
HotSpot/Server, HotSpot/Server running JRuby, GNU R, LLVM AOT compiled, V8

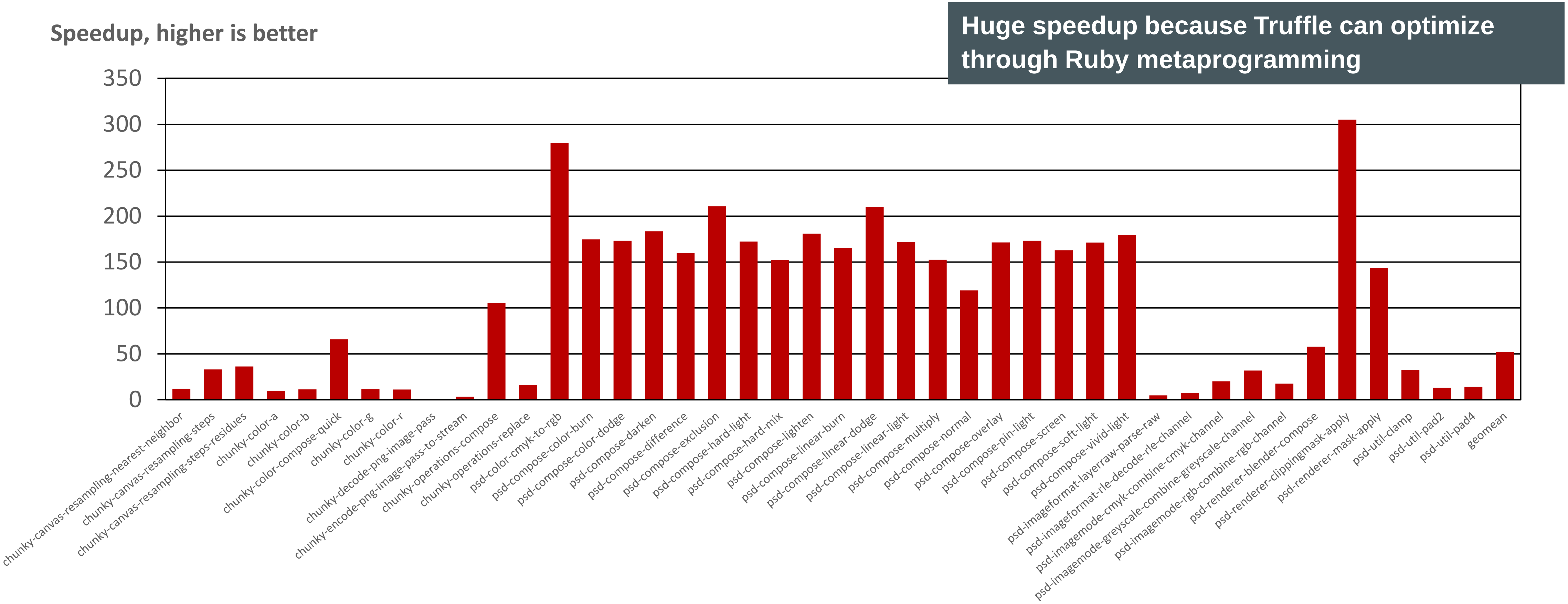
Performance: JavaScript

JavaScript performance: similar to V8



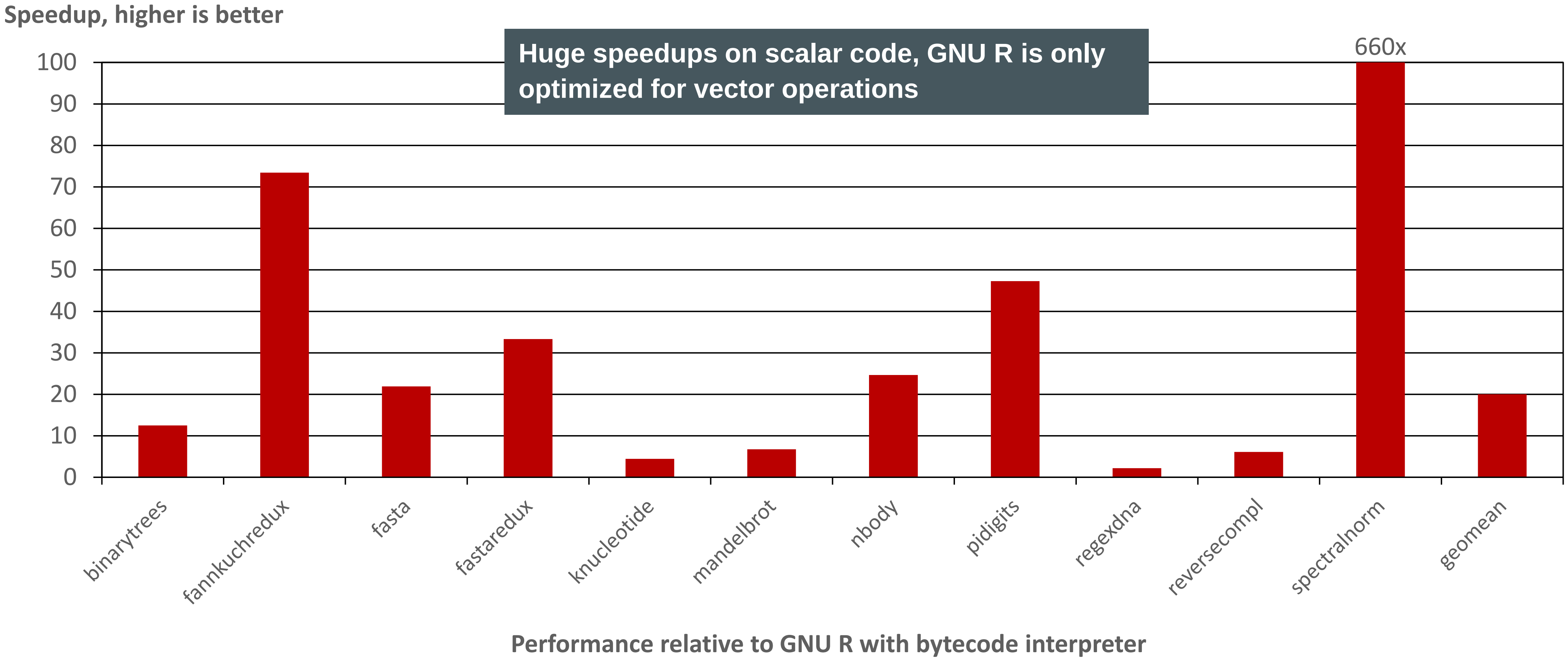
Performance: Ruby Compute-Intensive Kernels

Huge speedup because Truffle can optimize through Ruby metaprogramming



Performance relative to JRuby running with Java HotSpot server compiler

Performance: R with Scalar Code



Huge speedups on scalar code, GNU R is only optimized for vector operations

Acknowledgements

Oracle
Danilo Ansaloni
Stefan Anzinger
Cosmin Basca
Daniele Bonetta
Matthias Brantner
Petr Chalupa
Jürgen Christ
Laurent Daynès
Gilles Duboscq
Martin Entlicher
Bastian Hossbach
Christian Humer
Mick Jordan
Vojin Jovanovic
Peter Kessler
David Leopoldseder
Kevin Menard
Jakub Podlešák
Aleksandar Prokopec
Tom Rodriguez

Oracle (continued)
Roland Schatz
Chris Seaton
Doug Simon
Štěpán Šindelář
Zbyněk Šlajchrt
Lukas Stadler
Codrut Stancu
Jan Štola
Jaroslav Tulach
Michael Van De Vanter
Adam Welc
Christian Wimmer
Christian Wirth
Paul Wögerer
Mario Wolczko
Andreas Wöß
Thomas Würthinger

Oracle Interns
Brian Belleville
Miguel Garcia
Shams Imam
Alexey Karyakin
Stephen Kell
Andreas Kunft
Volker Lanting
Gero Leinemann
Julian Lettner
David Piorkowski
Gregor Richards
Robert Seilbeck
Rifat Shariyar

Oracle Alumni
Erik Eckstein
Michael Haupt
Christos Kotselidis
Hyunjin Lee
David Leibs
Chris Thalinger
Till Westmann

JKU Linz
Prof. Hanspeter Mössenböck
Benoit Daloze
Josef Eisl
Thomas Feichtinger
Matthias Grimmer
Christian Häubl
Josef Haider
Christian Huber
Stefan Marr
Manuel Rigger
Stefan Rumzucker
Bernhard Urban

University of Edinburgh
Christophe Dubach
Juan José Fumero Alfonso
Ranjeet Singh
Toomas Remmelg

LaBRI
Floréal Morandat

University of California, Irvine
Prof. Michael Franz
Gulfem Savrun Yeniceri
Wei Zhang

Purdue University
Prof. Jan Vitek
Tomas Kalibera
Petr Maj Lei Zhao

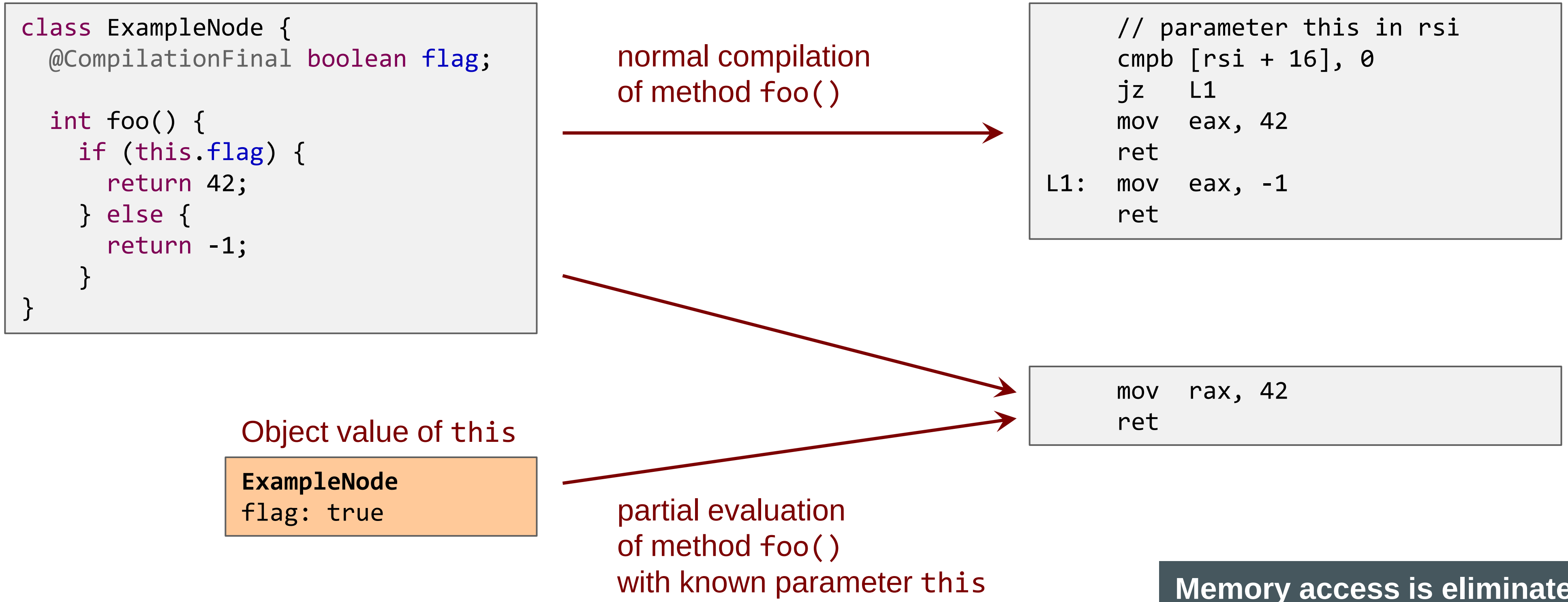
T. U. Dortmund
Prof. Peter Marwedel
Helena Kotthaus
Ingo Korb

University of California, Davis
Prof. Duncan Temple Lang
Nicholas Ulle

University of Lugano, Switzerland
Prof. Walter Binder
Sun Haiyang
Yudi Zheng

Partial Evaluation and Transfer to Interpreter

Example: Partial Evaluation



```
class ExampleNode {
  @CompilationFinal boolean flag;

  int foo() {
    if (this.flag) {
      return 42;
    } else {
      return -1;
    }
}
```

normal compilation
of method `foo()`

```
      // parameter this in rsi
      cmpb [rsi + 16], 0
      jz   L1
      mov  eax, 42
      ret
L1:   mov  eax, -1
      ret
```

Object value of `this`

```
ExampleNode
flag: true
```

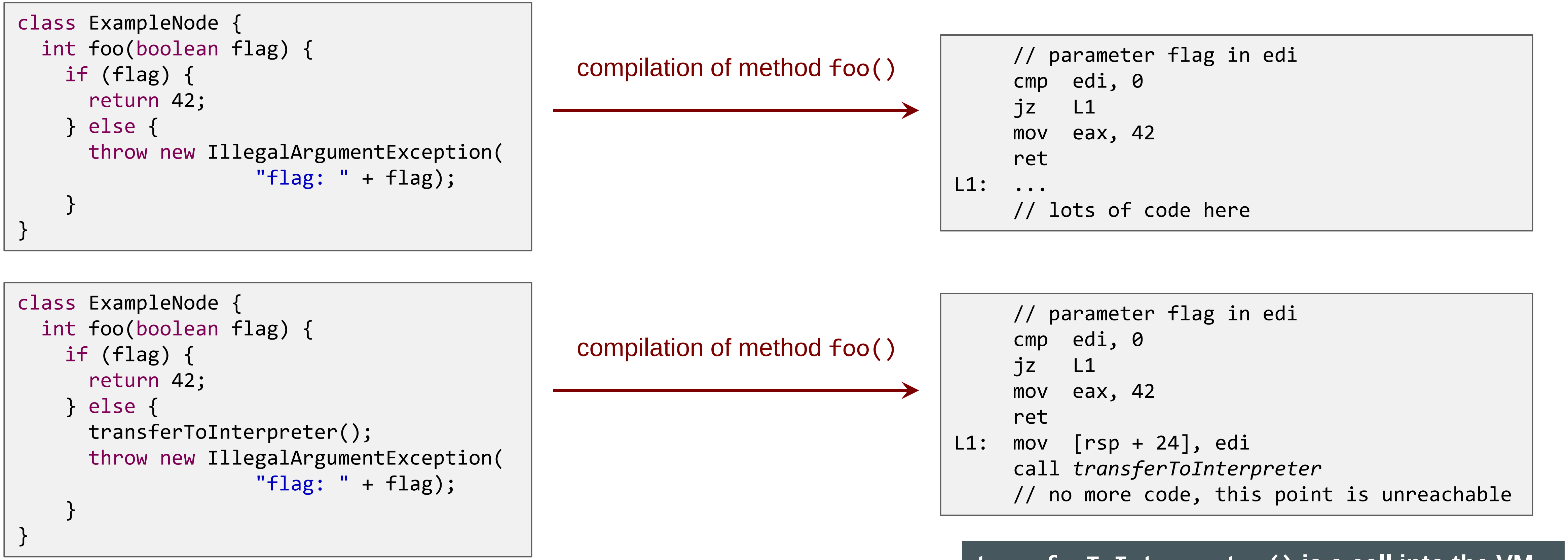
partial evaluation
of method `foo()`
with known parameter `this`

```
      mov  rax, 42
      ret
```

Memory access is eliminated and condition is constant folded during partial evaluation

`@CompilationFinal` field is treated like a `final` field during partial evaluation

Example: Transfer to Interpreter



```java
class ExampleNode {
  int foo(boolean flag) {
    if (flag) {
      return 42;
    } else {
      throw new IllegalArgumentException(
                  "flag: " + flag);
    }
}
```

compilation of method foo()

```
     // parameter flag in edi
     cmp  edi, 0
     jz   L1
     mov  eax, 42
     ret
L1:  ...
     // lots of code here
```

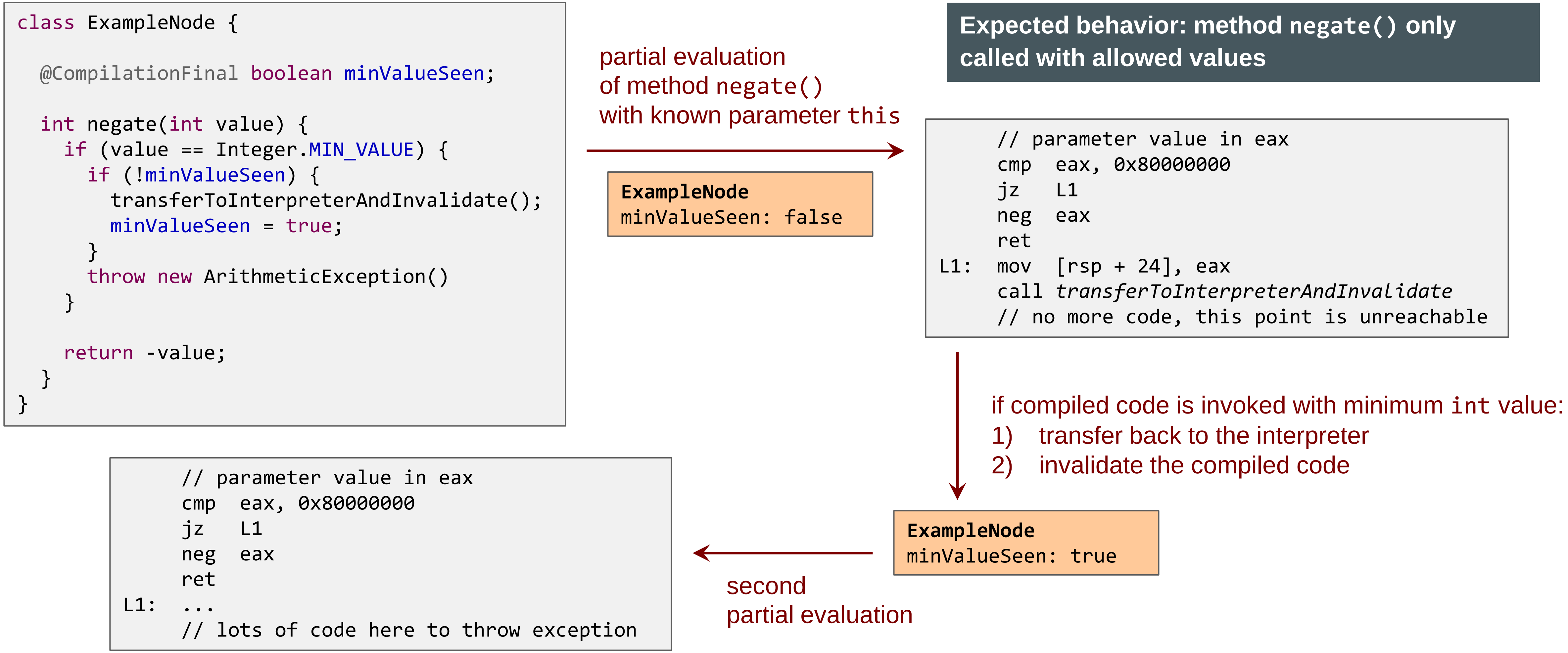
```java
class ExampleNode {
  int foo(boolean flag) {
    if (flag) {
      return 42;
    } else {
      transferToInterpreter();
      throw new IllegalArgumentException(
                  "flag: " + flag);
    }
}
```

compilation of method foo()

```
     // parameter flag in edi
     cmp  edi, 0
     jz   L1
     mov  eax, 42
     ret
L1:  mov  [rsp + 24], edi
     call transferToInterpreter
     // no more code, this point is unreachable
```

`transferToInterpreter()` is a call into the VM runtime that does not return to its caller, because execution continues in the interpreter

Example: Partial Evaluation and Transfer to Interpreter



```
class ExampleNode {

  @CompilationFinal boolean minValueSeen;

  int negate(int value) {
    if (value == Integer.MIN_VALUE) {
      if (!minValueSeen) {
        transferToInterpreterAndInvalidate();
        minValueSeen = true;
      }
      throw new ArithmeticException()
    }

    return -value;
  }
}
```

partial evaluation
of method `negate()`
with known parameter `this`

```
ExampleNode
minValueSeen: false
```

Expected behavior: method `negate()` only called with allowed values

```
     // parameter value in eax
     cmp   eax, 0x80000000
     jz    L1
     neg   eax
     ret
L1:  mov   [rsp + 24], eax
     call  transferToInterpreterAndInvalidate
     // no more code, this point is unreachable
```

if compiled code is invoked with minimum `int` value:

1) transfer back to the interpreter
2) invalidate the compiled code

```
ExampleNode
minValueSeen: true
```

second
partial evaluation

```
     // parameter value in eax
     cmp   eax, 0x80000000
     jz    L1
     neg   eax
     ret
L1:  ...
     // lots of code here to throw exception
```

Branch Profiles

```
class ExampleNode {

  final BranchProfile minValueSeen = BranchProfile.create();

  int negate(int value) {
    if (value == Integer.MIN_VALUE) {
      minValueSeen.enter();
      throw new ArithmeticException();
    }
    return -value;
  }
}
```

Truffle profile API provides high-level API that hides complexity and is easier to use

Best Practice: Use classes in `com.oracle.truffle.api.profiles` when possible, instead of `@CompilationFinal`

Condition Profiles for Branch Probability

```
class ExampleNode {

  final ConditionProfile positive = ConditionProfile.createCountingProfile();
  final BranchProfile minValueSeen = BranchProfile.create();

  int abs(int value) {
    if (positive.profile(value >= 0)) {
      return value;

    } else if (value == Integer.MIN_VALUE) {
      minValueSeen.enter();
      throw new ArithmeticException();

    } else {
      return -value;
    }
  }
}
```

Counting `ConditionProfile`: add branch probability for code paths with different execution frequencies

`BranchProfile`: remove unlikely code paths

Profiles: Summary

Profiles are for local speculation only (only invalidate one compiled method)

- `BranchProfile` to speculate on unlikely branches
  - Benefit: remove code of unlikely code paths
- `ConditionProfile` to speculate on conditions
  - `createBinaryProfile` does not profile probabilities
    - Benefit: remove code of unlikely branches
  - `createCountingProfile` profiles probabilities
    - Benefit: better machine code layout for branches with asymmetric execution frequency
- ValueProfile to speculate on Object values
  - createClassProfile to profile the class of the Object
    - Benefit: compiler has a known type for a value and can, e.g., replace virtual method calls with direct method calls and then inline the callee
  - createIdentityProfile to profile the object identity
    - Benefit: compiler has a known compile time constant Object value and can, e.g., constant fold final field loads
- PrimitiveValueProfile
  - Benefit: compiler has a known compile time constant primitive value an can, e.g., constant fold arithmetic operations

Assumptions

Create an assumption:

```
Assumption assumption = Truffle.getRuntime().createAssumption();
```

Assumptions allow non-local speculation (across multiple compiled methods)

Check an assumption:

```
void foo() {
  if (assumption.isValid()) {
    // Fast-path code that is only valid if assumption is true.
  } else {
    // Perform node specialization, or other slow-path code to respond to change.
  }
}
```

Checking an assumption does not need machine code, it really is a "free lunch"

Invalidate an assumption:

```
assumption.invalidate();
```

When an assumption is invalidate, all compiled methods that checked it are invalidated

Example: Assumptions

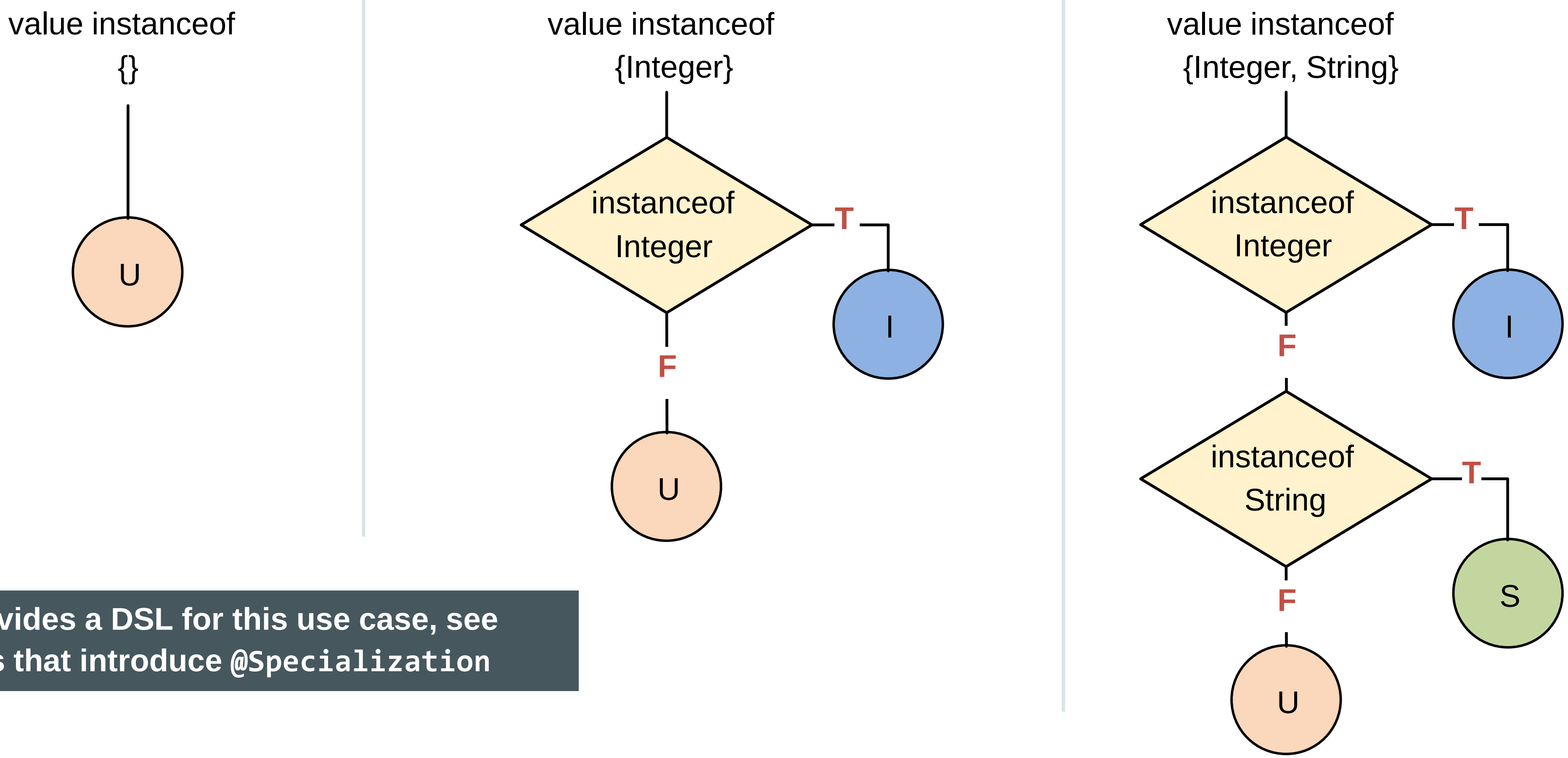
```
class ExampleNode {

  public static final Assumption addNotRedefined = Truffle.getRuntime().createAssumption();

  int add(int left, int right) {
    if (addNotRedefined.isValid()) {
      return left + right;
    } else {
      ...
      // Complicated code to call user-defined add function
    }
  }
}
```

Expected behavior: user does not redefine "+" for integer values

```
void redefineFunction(String name, ...) {
  if (name.equals("+")) {
    addNotRedefined.invalidate()) {
    ...
  }
}
```

This is not a synthetic example: Ruby allows redefinition of all operators on all types, including the standard numeric types

Specialization


value instanceof {}

value instanceof {Integer}

value instanceof {Integer, String}

Truffle provides a DSL for this use case, see later slides that introduce `@Specialization`

Profile, Assumption, or Specialization?

- Use profiles where local, monomorphic speculation is sufficient
  - Transfer to interpreter is triggered by the compiled method itself
  - Recompilation does not speculate again

- Use assumptions for non-local speculation
  - Transfer to interpreter is triggered from outside of a compiled method
  - Recompilation often speculates on a new assumption (or does not speculate again)

- Use specializations for local speculations where polymorphism is required
  - Transfer to interpreter is triggered by the compiled method method
  - Interpreter adds a new specialization
  - Recompilation speculates again, but with more allowed cases

A Simple Language

SL: A Simple Language

- Language to demonstrate and showcase features of Truffle
  - Simple and clean implementation
  - Not the language for your next implementation project

- Language highlights
  - Dynamically typed
  - Strongly typed
    - No automatic type conversions
  - Arbitrary precision integer numbers
  - First class functions
  - Dynamic function redefinition
  - Objects are key-value stores
    - Key and value can have any type, but typically the key is a String

About 2.5k lines of code

Types

SL Type	Values	Java Type in Implementation
Number	Arbitrary precision integer numbers	`long` for values that fit within 64 bits `java.lang.BigInteger` on overflow
Boolean	true, false	`boolean`
String	Unicode characters	`java.lang.String`
Function	Reference to a function	`SLFunction`
Object	key-value store	`DynamicObject`
Null	null	`SLNull.SINGLETON`

Null is its own type; could also be called "Undefined"

Best Practice: Use Java primitive types as much as possible to increase performance

Best Practice: Do not use the Java `null` value for the guest language null value

Syntax

- C-like syntax for control flow
  - `if`, `while`, `break`, `continue`, `return`
- Operators
  - +, -, *, /, ==, !=, <, <=, >, >=, &&, | |, ( )
  - + is defined on String, performs String concatenation
  - && and | | have short-circuit semantics
  - . or [] for property access
- Literals
  - Number, String, Function
- Builtin functions
  - println, readln: Standard I/O
  - nanoTime: to allow time measurements
  - defineFunction: dynamic function redefinition
  - stacktrace, helloEqualsWorld: stack walking and stack frame manipulation
  - new: Allocate a new object without properties

Parsing

- Scanner and parser generated from grammar
  - Using Coco/R
  - Available from http://ssw.jku.at/coco/

- Refer to Coco/R documentation for details
  - This is not a tutorial about parsing

- Building a Truffle AST from a parse tree is usually simple

Best Practice: Use your favorite parser generator, or an existing parser for your language

SL Examples

Hello World:

```
function main() {
  println("Hello World!");
}
```

```
Hello World!
```

Strings:

```
function f(a, b) {
  return a + " < " + b + ": " + (a < b);
}

function main() {
  println(f(2, 4));
  println(f(2, "4"));
}
```

```
2 < 4: true
Type error
```

Objects:

```
function main() {
  obj = new();
  obj.prop = "Hello World!";
  println(obj["pr" + "op"]);
}
```

```
Hello World!
```

Simple loop:

```
function main() {
  i = 0;
  sum = 0;
  while (i <= 10000) {
    sum = sum + i;
    i = i + 1;
  }
  return sum;
}
```

```
50005000
```

First class functions:

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

function foo(f) {
  println(f(40, 2));
}

function main() {
  foo(add);
  foo(sub);
}
```

```
42
38
```

Function definition and redefinition:

```
function foo() { println(f(40, 2)); }

function main() {
  defineFunction("function f(a, b) { return a + b; }");
  foo();

  defineFunction("function f(a, b) { return a - b; }");
  foo();
}
```

```
42
38
```

Getting Started

- Clone repository
  - `git clone https://github.com/graalvm/simplelanguage`

Version used in this tutorial: tag PLDI_2016

- Download Graal VM Development Kit
  - http://www.oracle.com/technetwork/oracle-labs/program-languages/downloads
  - Unpack the downloaded `graalvm_*.tar.gz` into `simplelanguage/graalvm`
  - Verify that launcher exists and is executable: `simplelanguage/graalvm/bin/java`

Version used in this tutorial: Graal VM 0.12

- Build
  - `mvn package`
- Run example program
  - `./sl tests/HelloWorld.sl`
- IDE Support
  - Import the Maven project into your favorite IDE
  - Instructions for Eclipse, NetBeans, IntelliJ are in README.md

ORACLE

Simple Tree Nodes

AST Interpreters

- AST = Abstract Syntax Tree
  - The tree produced by a parser of a high-level language compiler
- Every node can be executed
  - For our purposes, we implement nodes as a class hierarchy
  - Abstract `execute` method defined in Node base class
  - Execute overwritten in every subclass
- Children of an AST node produce input operand values
  - Example: AddNode to perform addition has two children: `left` and `right`
    - AddNode.execute first calls left.execute and right.execute to compute the operand values
    - Then peforms the addition and returns the result
  - Example: `IfNode` has three children: `condition`, `thenBranch`, `elseBranch`
    - `IfNode.execute` first calls `condition.execute` to compute the condition value
    - Based on the condition value, it either calls `thenBranch.execute` or `elseBranch.execute` (but never both of them)
- Textbook summary
  - Execution in an AST interpreter is slow (virtual call for every executed node)
  - But, easy to write and reason about; portable

Truffle Nodes and Trees

- Class Node: base class of all Truffle tree nodes
  - Management of parent and children
  - Replacement of this node with a (new) node
  - Copy a node
  - No execute() methods: define your own in subclasses
- Class NodeUtil provides useful utility methods

```
public abstract class Node implements Cloneable {

  public final Node getParent() { ... }
  public final Iterable<Node> getChildren() { ... }

  public final <T extends Node> T replace(T newNode) { ... }
  public Node copy() { ... }

  public SourceSection getSourceSection();
}
```

If Statement

```
public final class SLIfNode extends SLStatementNode {
  @Child private SLExpressionNode conditionNode;
  @Child private SLStatementNode thenPartNode;
  @Child private SLStatementNode elsePartNode;

  public SLIfNode(SLExpressionNode conditionNode, SLStatementNode thenPartNode, SLStatementNode elsePartNode) {
    this.conditionNode = conditionNode;
    this.thenPartNode = thenPartNode;
    this.elsePartNode = elsePartNode;
  }

  public void executeVoid(VirtualFrame frame) {
    if (conditionNode.executeBoolean(frame)) {
      thenPartNode.executeVoid(frame);
    } else {
      elsePartNode.executeVoid(frame);
    }
  }
}
```

Rule: A field for a child node must be annotated with `@Child` and must not be `final`

If Statement with Profiling

```
public final class SLIfNode extends SLStatementNode {
  @Child private SLExpressionNode conditionNode;
  @Child private SLStatementNode thenPartNode;
  @Child private SLStatementNode elsePartNode;

  private final ConditionProfile condition = ConditionProfile.createCountingProfile();

  public SLIfNode(SLExpressionNode conditionNode, SLStatementNode thenPartNode, SLStatementNode elsePartNode) {
    this.conditionNode = conditionNode;
    this.thenPartNode = thenPartNode;
    this.elsePartNode = elsePartNode;
  }

  public void executeVoid(VirtualFrame frame) {
    if (condition.profile(conditionNode.executeBoolean(frame))) {
      thenPartNode.executeVoid(frame);
    } else {
      elsePartNode.executeVoid(frame);
    }
  }
}
```

Best practice: Profiling in the interpreter allows the compiler to generate better code

Blocks

```
public final class SLBlockNode extends SLStatementNode {
  @Children private final SLStatementNode[] bodyNodes;

  public SLBlockNode(SLStatementNode[] bodyNodes) {
    this.bodyNodes = bodyNodes;
  }

  @ExplodeLoop
  public void executeVoid(VirtualFrame frame) {
    for (SLStatementNode statement : bodyNodes) {
      statement.executeVoid(frame);
    }
  }
}
```

Rule: A field for multiple child nodes must be annotated with `@Children` and a `final` array

Rule: The iteration of the children must be annotated with `@ExplodeLoop`

Return Statement: Inter-Node Control Flow

```
public final class SLReturnNode extends SLStatementNode {
  @Child private SLExpressionNode valueNode;
  ...
  public void executeVoid(VirtualFrame frame) {
    throw new SLReturnException(valueNode.executeGeneric(frame));
  }
}
```

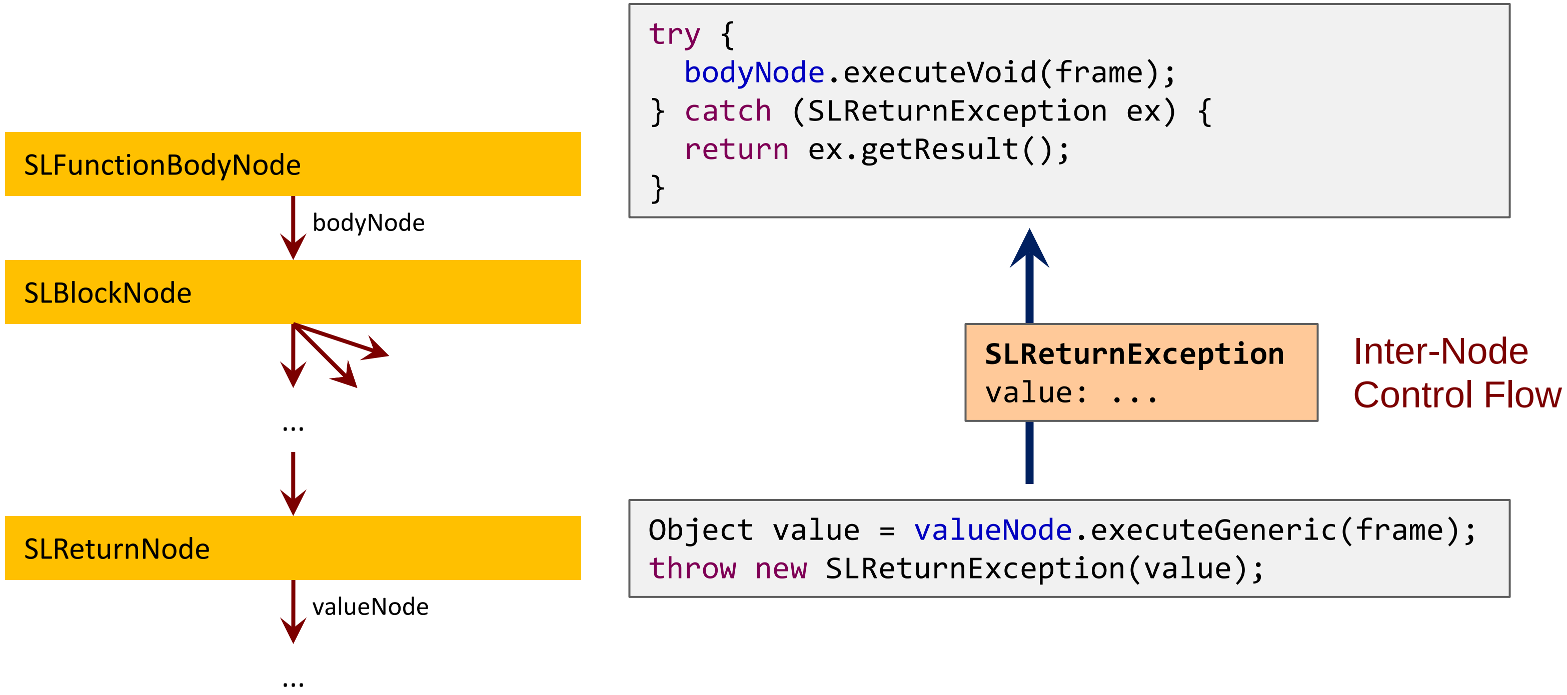
```
public final class SLFunctionBodyNode extends SLExpressionNode {
  @Child private SLStatementNode bodyNode;
  ...
  public Object executeGeneric(VirtualFrame frame) {
    try {
      bodyNode.executeVoid(frame);
    } catch (SLReturnException ex) {
      return ex.getResult();
    }
    return SLNull.SINGLETON;
  }
}
```

```
public final class SLReturnException
      extends ControlFlowException {

  private final Object result;
  ...
}
```

Best practice: Use Java exceptions for inter-node control flow

Rule: Exceptions used to model control flow extend `ControlFlowException`

Exceptions for Inter-Node Control Flow



SLFunctionBodyNode

bodyNode

SLBlockNode

...

SLReturnNode

valueNode

...

```
try {
  bodyNode.executeVoid(frame);
} catch (SLReturnException ex) {
  return ex.getResult();
}
```

SLReturnException
value: ...

Inter-Node
Control Flow

```
Object value = valueNode.executeGeneric(frame);
throw new SLReturnException(value);
```

Exception unwinds all the interpreter stack frames of the method (loops, conditions, blocks, ...)

Truffle DSL for Specializations

Addition

```
@NodeChildren({@NodeChild("leftNode"), @NodeChild("rightNode")})
public abstract class SLBinaryNode extends SLExpressionNode { }

public abstract class SLAddNode extends SLBinaryNode {

  @Specialization(rewriteOn = ArithmeticException.class)
  protected final long add(long left, long right) {
    return ExactMath.addExact(left, right);
  }

  @Specialization
  protected final BigInteger add(BigInteger left, BigInteger right) {
    return left.add(right);
  }

  @Specialization(guards = "isString(left, right)")
  protected final String add(Object left, Object right) {
    return left.toString() + right.toString();
  }

  protected final boolean isString(Object a, Object b) {
    return a instanceof String || b instanceof String;
  }
}
```

The order of the @Specialization methods is important: the first matching specialization is selected

For all other specializations, guards are implicit based on method signature

ORACLE

Code Generated by Truffle DSL (1)

Generated code with factory method:

```
@GeneratedBy(SLAddNode.class)
public final class SLAddNodeGen extends SLAddNode {

  public static SLAddNode create(SLExpressionNode leftNode, SLExpressionNode rightNode) { ... }

  ...
}
```

The parser uses the factory to create a node that is initially in the uninitialized state

The generated code performs all the transitions between specialization states

Code Generated by Truffle DSL (2)

```
@GeneratedBy(methodName = "add(long, long)", value = SLAddNode.class)
private static final class Add0Node_ extends BaseNode_ {
  @Override
  public long executeLong(VirtualFrame frameValue) throws UnexpectedResultException {
    long leftNodeValue_;
    try {
      leftNodeValue_ = root.leftNode_.executeLong(frameValue);
    } catch (UnexpectedResultException ex) {
      Object rightNodeValue = executeRightNode_(frameValue);
      return SLTypesGen.expectLong(getNext().execute_(frameValue, ex.getResult(), rightNodeValue));
    }
    long rightNodeValue_;
    try {
      rightNodeValue_ = root.rightNode_.executeLong(frameValue);
    } catch (UnexpectedResultException ex) {
      return SLTypesGen.expectLong(getNext().execute_(frameValue, leftNodeValue_, ex.getResult()));
    }
    try {
      return root.add(leftNodeValue_, rightNodeValue_);
    } catch (ArithmeticException ex) {
      root.excludeAdd0_ = true;
      return SLTypesGen.expectLong(remove("threw rewrite exception", frameValue, leftNodeValue_, rightNodeValue_));
    }
  }

  @Override
  public Object execute(VirtualFrame frameValue) {
    try {
      return executeLong(frameValue);
    } catch (UnexpectedResultException ex) {
      return ex.getResult();
    }
}
```

The generated code can and will change at any time

Type System Definition in Truffle DSL

```
@TypeSystem({long.class, BigInteger.class, boolean.class,
             String.class, SLFunction.class, SLNull.class})

public abstract class SLTypes {
  @ImplicitCast
  public BigInteger castBigInteger(long value) {
    return BigInteger.valueOf(value);
  }
}
```

Not shown in slide: Use `@TypeCheck` and `@TypeCast` to customize type conversions

```
@TypeSystemReference(SLTypes.class)
public abstract class SLExpressionNode extends SLStatementNode {

  public abstract Object executeGeneric(VirtualFrame frame);

  public long executeLong(VirtualFrame frame) throws UnexpectedResultException {
    return SLTypesGen.SLTYPES.expectLong(executeGeneric(frame));
  }
  public boolean executeBoolean(VirtualFrame frame) ...
}
```

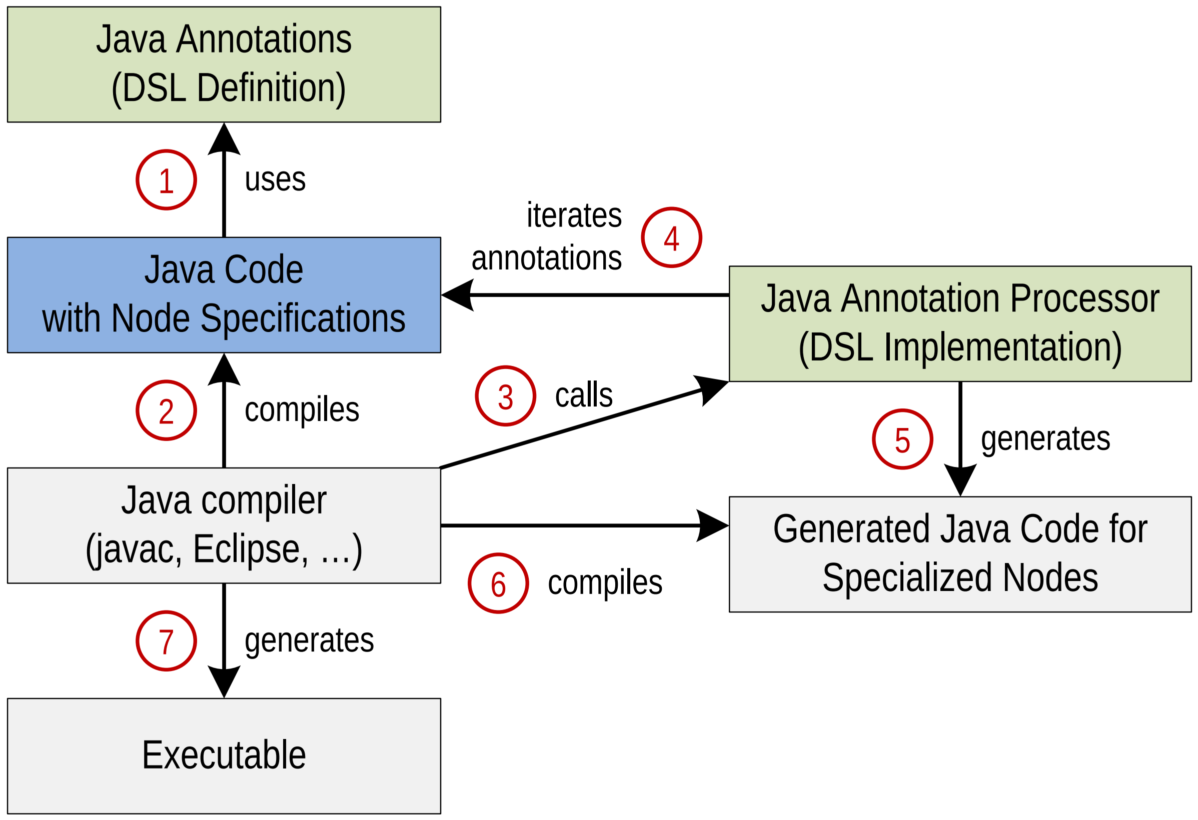
SLTypesGen is a generated subclass of SLTypes

Rule: One `execute()` method per type you want to specialize on, in addition to the abstract `executeGeneric()` method

`UnexpectedResultException`

- Type-specialized `execute()` methods have specialized return type
  - Allows primitive return types, to avoid boxing
  - Allows to use the result without type casts
  - Speculation types are stable and the specialization fits
- But what to do when speculation was too optimistic?
  - Need to return a value with a type more general than the return type
  - Solution: return the value “boxed” in an `UnexpectedResultException`
- Exception handler performs node rewriting
  - Exception is thrown only once, so no performance bottleneck

Truffle DSL Workflow


Java Annotations (DSL Definition)
1 uses
Java Code with Node Specifications
iterates annotations 4
Java Annotation Processor (DSL Implementation)
2 compiles
3 calls
5 generates
Java compiler (javac, Eclipse, …)
Generated Java Code for Specialized Nodes
6 compiles
7 generates
Executable

Frames and Local Variables

Frame Layout

- In the interpreter, a frame is an object on the heap
  - Allocated in the function prologue
  - Passed around as parameter to `execute()` methods
- The compiler eliminates the allocation
  - No object allocation and object access
  - Guest language local variables have the same performance as Java local variables
- `FrameDescriptor`: describes the layout of a frame
  - A mapping from identifiers (usually variable names) to typed slots
  - Every slot has a unique index into the frame object
  - Created and filled during parsing
- `Frame`
  - Created for every invoked guest language function

Frame Management

- Truffle API only exposes frame interfaces
  - Implementation class depends on the optimizing system
- `VirtualFrame`
  - What you usually use: automatically optimized by the compiler
  - Must never be assigned to a field, or escape out of an interpreted function
- `MaterializedFrame`
  - A frame that can be stored without restrictions
  - Example: frame of a closure that needs to be passed to other function
- Allocation of frames
  - Factory methods in the class `TruffleRuntime`

Frame Management

```java
public interface Frame {
  FrameDescriptor getFrameDescriptor();
  Object[] getArguments();

  boolean isType(FrameSlot slot);
  Type getType(FrameSlot slot) throws FrameSlotTypeException;
  void setType(FrameSlot slot, Type value);

  Object getValue(FrameSlot slot);

  MaterializedFrame materialize();
}
```

Frames support all Java primitive types, and `Object`

SL types String, SLFunction, and SLNull are stored as `Object` in the frame

Rule: Never allocate frames yourself, and never make your own frame implementations

Local Variables

```
@NodeChild("valueNode")
@NodeField(name = "slot", type = FrameSlot.class)
public abstract class SLWriteLocalVariableNode extends SLExpressionNode {

  protected abstract FrameSlot getSlot();

  @Specialization(guards = "isLongOrIllegal(frame)")
  protected long writeLong(VirtualFrame frame, long value) {
    getSlot().setKind(FrameSlotKind.Long);
    frame.setLong(getSlot(), value);
    return value;
  }
  protected boolean isLongOrIllegal(VirtualFrame frame) {
    return getSlot().getKind() == FrameSlotKind.Long || getSlot().getKind() == FrameSlotKind.Illegal;
  }
  ...

  @Specialization(contains = {"writeLong", "writeBoolean"})
  protected Object write(VirtualFrame frame, Object value) {
    getSlot().setKind(FrameSlotKind.Object);
    frame.setObject(getSlot(), value);
    return value;
  }
}
```

`setKind()` is a no-op if kind is already `Long`

If we cannot specialize on a single primitive type, we switch to Object for all reads and writes

ORACLE

Local Variables

```
@NodeField(name = "slot", type = FrameSlot.class)
public abstract class SLReadLocalVariableNode extends SLExpressionNode {

  protected abstract FrameSlot getSlot();

  @Specialization(guards = "isLong(frame)")
  protected long readLong(VirtualFrame frame) {
    return FrameUtil.getLongSafe(frame, getSlot());
  }
  protected boolean isLong(VirtualFrame frame) {
    return getSlot().getKind() == FrameSlotKind.Long;
  }
  ...

  @Specialization(contains = {"readLong", "readBoolean"})
  protected Object readObject(VirtualFrame frame) {
    if (!frame.isObject(getSlot())) {
      CompilerDirectives.transferToInterpreter();
      Object result = frame.getValue(getSlot());
      frame.setObject(getSlot(), result);
      return result;
    }

    return FrameUtil.getObjectSafe(frame, getSlot());
  }
```

The guard ensure the frame slot contains a primitive long value

Slow path: we can still have frames with primitive values written before we switched the local variable to the kind Object

Compilation

Compilation

- Automatic partial evaluation of AST
  - Automatically triggered by function execution count

- Compilation assumes that the AST is stable
  - All `@Child` and `@Children` fields treated like `final` fields
- Later node rewriting invalidates the machine code
  - Transfer back to the interpreter: “Deoptimization”
  - Complex logic for node rewriting not part of compiled code
  - Essential for excellent peak performance

- Compiler optimizations eliminate the interpreter overhead
  - No more dispatch between nodes
  - No more allocation of `VirtualFrame` objects
  - No more exceptions for inter-node control flow

Truffle Compilation API

- Default behavior of compilation: Inline all reachable Java methods

- Truffle API provides class `CompilerDirectives` to influence compilation
  - `@CompilationFinal`
    - Treat a field as `final` during compilation
  - `transferToInterpreter()`
    - Never compile part of a Java method
  - `transferToInterpreterAndInvalidate()`
    - Invalidate machine code when reached
    - Implicitly done by `Node.replace()`
  - `@TruffleBoundary`
    - Marks a method that is not important for performance, i.e., not part of partial evaluation
  - `inInterpreter()`
    - For profiling code that runs only in the interpreter
  - `Assumption`
    - Invalidate machine code from outside
    - Avoid checking a condition over and over in compiled code

Slow Path Annotation

```java
public abstract class SLPrintlnBuiltin extends SLBuiltinNode {

  @Specialization
  public final Object println(Object value) {
    doPrint(getContext().getOutput(), value);
    return value;
  }

  @TruffleBoundary
  private static void doPrint(PrintStream out, Object value) {
    out.println(value);
  }
}
```

When compiling, the output stream is a constant

Why `@TruffleBoundary`? Inlining something as big as `println()` would lead to code explosion

Compiler Assertions

- You work hard to help the compiler
- How do you check that you succeeded?

- CompilerAsserts.partialEvaluationConstant()
  - Checks that the passed in value is a compile-time constant early during partial evaluation
- CompilerAsserts.compilationConstant()
  - Checks that the passed in value is a compile-time constant (not as strict as partialEvaluationConstant)
  - Compiler fails with a compilation error if the value is not a constant
  - When the assertion holds, no code is generated to produce the value
- CompilerAsserts.neverPartOfCompilation()
  - Checks that this code is never reached in a compiled method
  - Compiler fails with a compilation error if code is reachable
  - Useful at the beginning of helper methods that are big or rewrite nodes
  - All code dominated by the assertion is never compiled

Compilation

SL source code:

```
function loop(n) {
  i = 0;
  sum = 0;
  while (i <= n) {
    sum = sum + i;
    i = i + 1;
  }
  return sum;
}
```

Machine code for loop:

```
     mov  r14, 0
     mov  r13, 0
     jmp  L2
L1:  safepoint
     mov  rax, r13
     add  rax, r14
     jo   L3
     inc  r13
     mov  r14, rax
L2:  cmp  r13, rbp
     jle  L1
     ...
L3:  call transferToInterpreter
```

Run this example:

```
./sl -dump -disassemble tests/SumPrint.sl
```

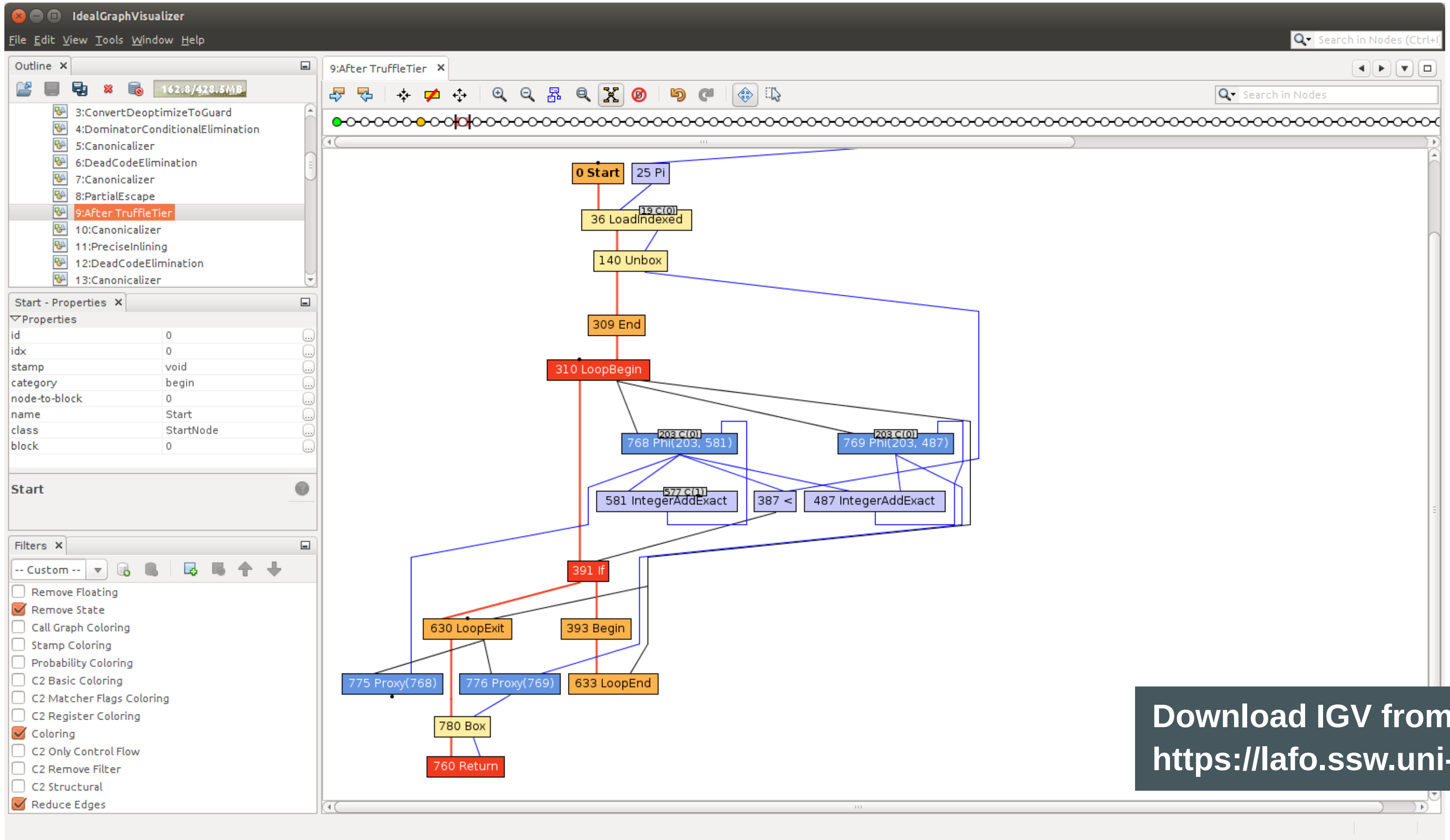
Truffle compilation printing is enabled

Background compilation is disabled

Graph dumping to IGV is enabled

Disassembling is enabled

Visualization Tools: IGV



Outline

162.8/428.5MB

- 3:ConvertDeoptimizeToGuard
- 4:DominatorConditionalElimination
- 5:Canonicalizer
- 6:DeadCodeElimination
- 7:Canonicalizer
- 8:PartialEscape
- 9:After TruffleTier
- 10:Canonicalizer
- 11:PreciseInlining
- 12:DeadCodeElimination
- 13:Canonicalizer

Start - Properties

Properties

id	0
idx	0
stamp	void
category	begin
node-to-block	0
name	Start
class	StartNode
block	0

Start

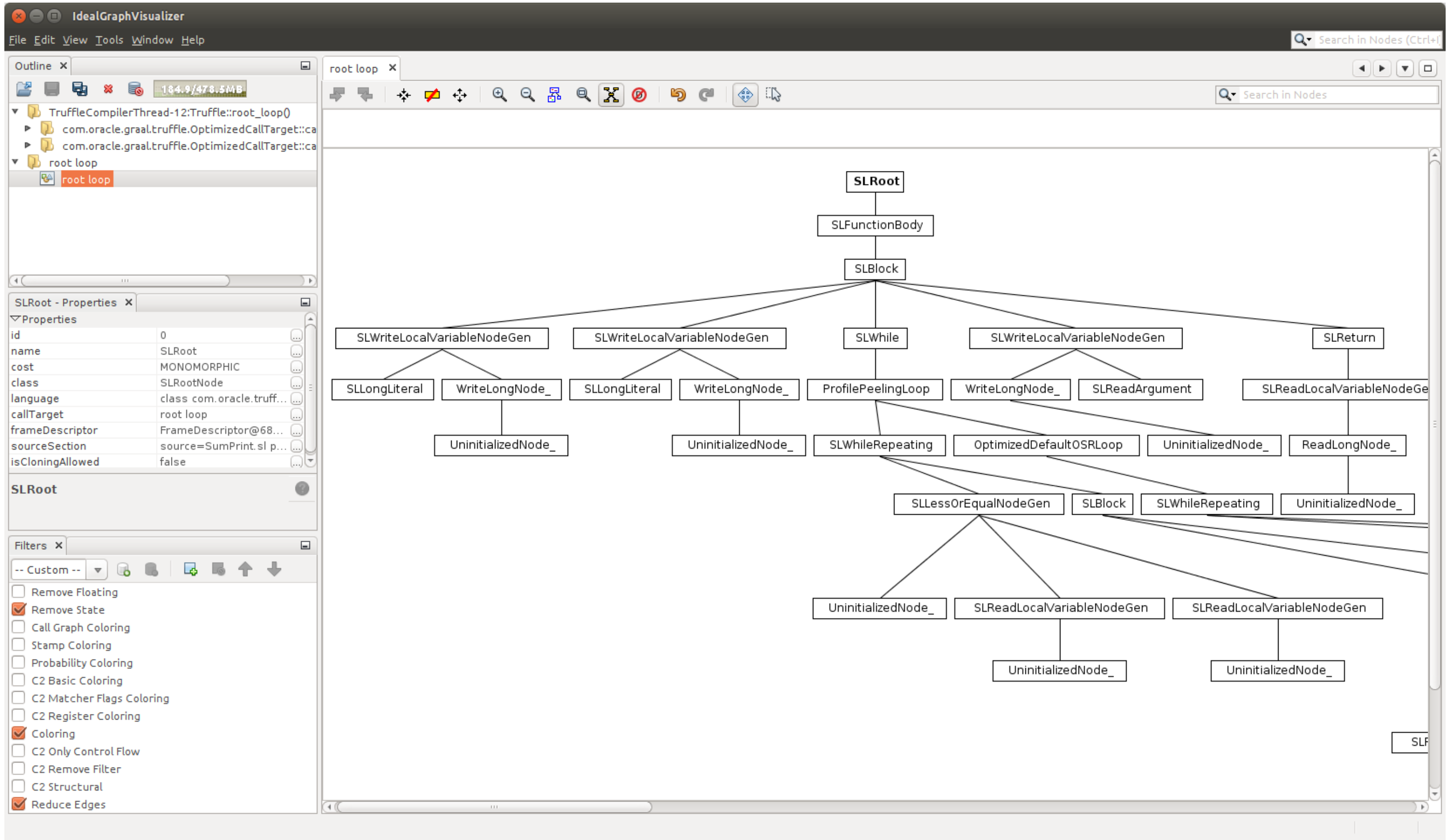
Filters

-- Custom --

- ☐ Remove Floating
- ☑ Remove State
- ☐ Call Graph Coloring
- ☐ Stamp Coloring
- ☐ Probability Coloring
- ☐ C2 Basic Coloring
- ☐ C2 Matcher Flags Coloring
- ☐ C2 Register Coloring
- ☑ Coloring
- ☐ C2 Only Control Flow
- ☐ C2 Remove Filter
- ☐ C2 Structural
- ☑ Reduce Edges

Download IGV from https://lafo.ssw.uni-linz.ac.at/pub/idealgraphvisualizer

Visualization Tools: IGV



- TruffleCompilerThread-12:Truffle::root_loop()
  - com.oracle.graal.truffle.OptimizedCallTarget::ca
  - com.oracle.graal.truffle.OptimizedCallTarget::ca
- root loop
  - root loop

SLRoot - Properties

Properties

id	0
name	SLRoot
cost	MONOMORPHIC
class	SLRootNode
language	class com.oracle.truff...
callTarget	root loop
frameDescriptor	FrameDescriptor@68...
sourceSection	source=SumPrint.sl p...
isCloningAllowed	false

SLRoot

Filters

-- Custom --

☐ Remove Floating
☑ Remove State
☐ Call Graph Coloring
☐ Stamp Coloring
☐ Probability Coloring
☐ C2 Basic Coloring
☐ C2 Matcher Flags Coloring
☐ C2 Register Coloring
☑ Coloring
☐ C2 Only Control Flow
☐ C2 Remove Filter
☐ C2 Structural
☑ Reduce Edges

Truffle Mindset

- Do not optimize interpreter performance
  - Only optimize compiled code performance
- Collect profiling information in interpreter
  - Yes, it makes the interpreter slower
  - But it makes your compiled code faster
- Do not specialize nodes in the parser, e.g., via static analysis
  - Trust the specialization at run time
- Keep node implementations small and simple
  - Split complex control flow into multiple nodes, use node rewriting
- Use `final` fields
  - Compiler can aggressively optimize them
  - Example: An `if` on a `final` field is optimized away by the compiler
  - Use profiles or `@CompilationFinal` if the Java `final` is too restrictive
- Use microbenchmarks to assess and track performance of specializations
  - Ensure and assert that you end up in the expected specialization

Truffle Mindset: Frames

- Use `VirtualFrame`, and ensure it does not escape
  - Graal must be able to inline all methods that get the `VirtualFrame` parameter
  - Call must be statically bound during compilation
  - Calls to `static` or `private` methods are always statically bound
  - Virtual calls and interface calls work if either
    - The receiver has a known exact type, e.g., comes from a `final` field
    - The method is not overridden in a subclass

- Important rules on passing around a `VirtualFrame`
  - Never assign it to a field
  - Never pass it to a recursive method
    - Graal cannot inline a call to a recursive method

- Use a `MaterializedFrame` if a `VirtualFrame` is too restrictive
  - But keep in mind that access is slower

Function Calls

Polymorphic Inline Caches

- Function lookups are expensive
  - At least in a real language, in SL lookups are only a few field loads
- Checking whether a function is the correct one is cheap
  - Always a single comparison
- Inline Cache
  - Cache the result of the previous lookup and check that it is still correct
- Polymorphic Inline Cache
  - Cache multiple previous lookups, up to a certain limit
- Inline cache miss needs to perform the slow lookup
- Implementation using tree specialization
  - Build chain of multiple cached functions

Example: Simple Polymorphic Inline Cache

```
public abstract class ANode extends Node {

    public abstract Object execute(Object operand);

    @Specialization(limit = "3",
                    guards = "operand == cachedOperand")
    protected Object doCached(AType operand,
                    @Cached("operand") AType cachedOperand) {
        // implementation
        return cachedOperand;
    }

    @Specialization(contains = "doCached")
    protected Object doGeneric(AType operand) {
        // implementation
        return operand;
    }
}
```

The `cachedOperand` is a compile time constant

Up to 3 compile time constants are cached

The generic case contains all cached cases, so the 4th unique value removes the cache chain

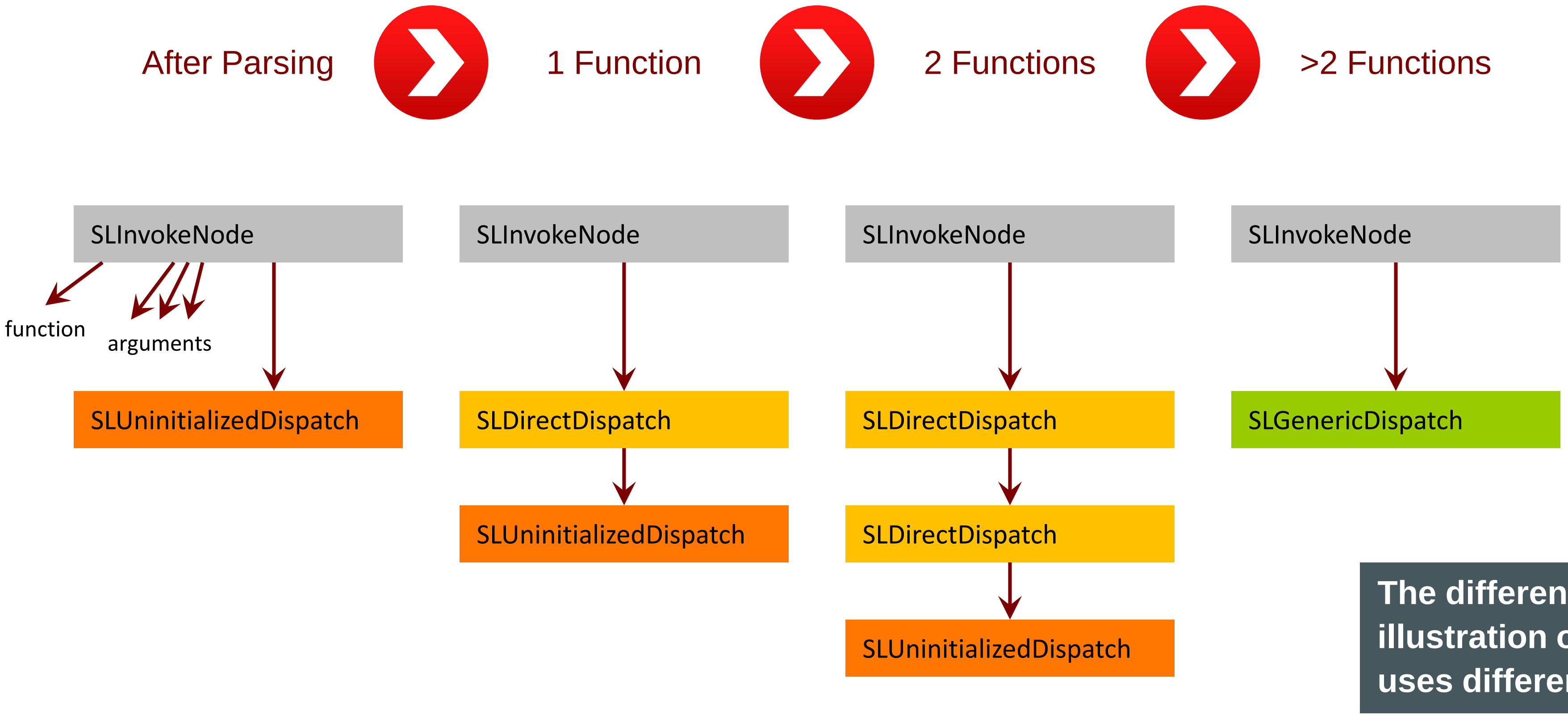
The `operand` is no longer a compile time constant

The `@Cached` annotation leads to a `final` field in the generated code

Compile time constants are usually the starting point for more constant folding

Polymorphic Inline Cache for Function Dispatch

Example of cache with length 2



After Parsing

SLInvokeNode

function

arguments

SLUninitializedDispatch

1 Function

SLInvokeNode

SLDirectDispatch

SLUninitializedDispatch

2 Functions

SLInvokeNode

SLDirectDispatch

SLDirectDispatch

SLUninitializedDispatch

>2 Functions

SLInvokeNode

SLGenericDispatch

The different dispatch nodes are for illustration only, the generated code uses different names

Invoke Node

```
public final class SLInvokeNode extends SLExpressionNode {

  @Child private SLExpressionNode functionNode;
  @Children private final SLExpressionNode[] argumentNodes;
  @Child private SLDispatchNode dispatchNode;

  @ExplodeLoop
  public Object executeGeneric(VirtualFrame frame) {
    Object function = functionNode.executeGeneric(frame);

    Object[] argumentValues = new Object[argumentNodes.length];
    for (int i = 0; i < argumentNodes.length; i++) {
      argumentValues[i] = argumentNodes[i].executeGeneric(frame);
    }

    return dispatchNode.executeDispatch(frame, function, argumentValues);
  }
}
```

Separation of concerns: this node evaluates the function and arguments only

Dispatch Node

```java
public abstract class SLDispatchNode extends Node {

  public abstract Object executeDispatch(VirtualFrame frame, Object function, Object[] arguments);

  @Specialization(limit = "2",
                  guards = "function == cachedFunction",
                  assumptions = "cachedFunction.getCallTargetStable()")
  protected static Object doDirect(VirtualFrame frame, SLFunction function, Object[] arguments,
                  @Cached("function") SLFunction cachedFunction,
                  @Cached("create(cachedFunction.getCallTarget())") DirectCallNode callNode) {

    return callNode.call(frame, arguments);
  }

  @Specialization(contains = "doDirect")
  protected static Object doIndirect(VirtualFrame frame, SLFunction function, Object[] arguments,
                  @Cached("create()") IndirectCallNode callNode) {

    return callNode.call(frame, function.getCallTarget(), arguments);
  }
}
```

Separation of concerns: this node builds the inline cache chain

ORACLE

Code Created from Guards and @Cached Parameters

Code creating the `doDirect` inline cache (runs infrequently):

```
if (number of doDirect inline cache entries < 2) {

if (function instanceof SLFunction) {

cachedFunction = (SLFunction) function;

~~if (function == cachedFunction) {~~

callNode = DirectCallNode.create(cachedFunction.getCallTarget());

assumption1 = cachedFunction.getCallTargetStable();

if (assumption1.isValid()) {

create and add new doDirect inline cache entry
```

Code checking the inline cache (runs frequently):

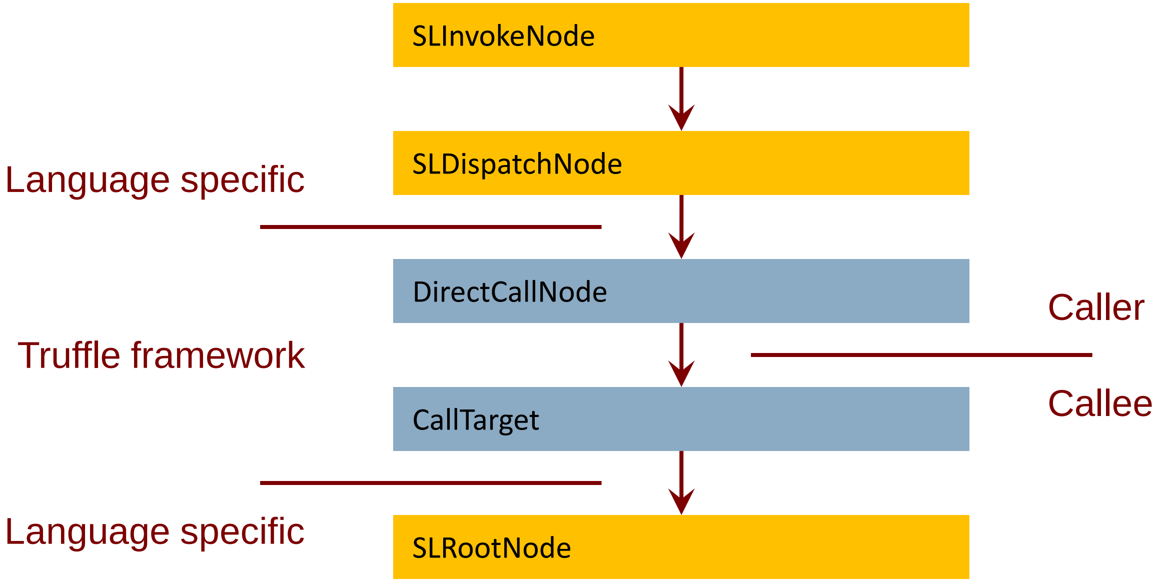
```
~~assumption1.check();~~

~~if (function instanceof SLFunction) {~~

if (function == cachedFunction)) {

callNode.call(frame, arguments);
```

Code that is compiled to a no-op is marked ~~strikethrough~~

The inline cache check is only one comparison with a compile time constant

Partial evaluation can go across function boundary (function inlining) because `callNode` with its `callTarget` is `final`

Language Nodes vs. Truffle Framework Nodes



Truffle framework code triggers compilation, function inlining, …

Function Redefinition (1)

- Problem
  - In SL, functions can be redefined at any time
  - This invalidates optimized call dispatch, and function inlining
  - Checking for redefinition before each call would be a huge overhead

- Solution
  - Every `SLFunction` has an `Assumption`
  - `Assumption` is invalidated when the function is redefined
    - This invalidates optimized machine code

- Result
  - No overhead when calling a function

Function Redefinition (2)

```
public abstract class SLDefineFunctionBuiltin extends SLBuiltinNode {

  @TruffleBoundary
  @Specialization
  public String defineFunction(String code) {
    Source source = Source.fromText(code, "[defineFunction]");
    getContext().getFunctionRegistry().register(Parser.parseSL(source));
    return code;
  }
}
```

Why `@TruffleBoundary`? Inlining something as big as the parser would lead to code explosion

SL semantics: Functions can be defined and redefined at any time

Function Redefinition (3)

```java
public final class SLFunction {

  private final String name;
  private RootCallTarget callTarget;
  private Assumption callTargetStable;

  protected SLFunction(String name) {
    this.name = name;
    this.callTarget = Truffle.getRuntime().createCallTarget(new SLUndefinedFunctionRootNode(name));
    this.callTargetStable = Truffle.getRuntime().createAssumption(name);
  }

  protected void setCallTarget(RootCallTarget callTarget) {
    this.callTarget = callTarget;
    this.callTargetStable.invalidate();
    this.callTargetStable = Truffle.getRuntime().createAssumption(name);
  }
}
```

The utility class `CyclicAssumption` simplifies this code

Function Arguments

- Function arguments are not type-specialized
  - Passed in `Object[]` array
- Function prologue writes them to local variables
  - `SLReadArgumentNode` in the function prologue
  - Local variable accesses are type-specialized, so only one unboxing

Example SL code:

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

function main() {
  add(2, 3);
}
```

Specialized AST for function `add()`:

```
SLRootNode
  bodyNode = SLFunctionBodyNode
    bodyNode = SLBlockNode
      bodyNodes[0] = SLWriteLocalVariableNode<writeLong>(name = "a")
        valueNode = SLReadArgumentNode(index = 0)
      bodyNodes[1] = SLWriteLocalVariableNode<writeLong>(name = "b")
        valueNode = SLReadArgumentNode(index = 1)
      bodyNodes[2] = SLReturnNode
        valueNode = SLAddNode<addLong>
          leftNode = SLReadLocalVariableNode<readLong>(name = "a")
          rightNode = SLReadLocalVariableNode<readLong>(name = "b")
```

Function Inlining vs. Function Splitting

- Function inlining is one of the most important optimizations
  - Replace a call with a copy of the callee
- Function inlining in Truffle operates on the AST level
  - Partial evaluation does not stop at `DirectCallNode`, but continues into next `CallTarget`
  - All later optimizations see the big combined tree, without further work
- Function splitting creates a new, uninitialized copy of an AST
  - Specialization in the context of a particular caller
  - Useful to avoid polymorphic specializations and to keep polymorphic inline caches shorter
  - Function inlining can inline a better specialized AST
  - Result: context sensitive profiling information
- Function inlining and function splitting are language independent
  - The Truffle framework is doing it automatically for you

ORACLE

Compilation with Inlined Function

SL source code without call:

```
function loop(n) {
  i = 0;
  sum = 0;
  while (i <= n) {
    sum = sum + i;
    i = i + 1;
  }
  return sum;
}
```

Machine code for loop without call:

```
     mov  r14, 0
     mov  r13, 0
     jmp  L2
L1:  safepoint
     mov  rax, r13
     add  rax, r14
     jo   L3
     inc  r13
     mov  r14, rax
L2:  cmp  r13, rbp
     jle  L1
     ...
L3:  call transferToInterpreter
```

SL source code with call:

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

function loop(n) {
  i = 0;
  sum = 0;
  while (i <= n) {
    sum = add(sum, i);
    i = add(i, 1);
  }
  return sum;
}
```

Machine code for loop with call:

```
     mov  r14, 0
     mov  r13, 0
     jmp  L2
L1:  safepoint
     mov  rax, r13
     add  rax, r14
     jo   L3
     inc  r13
     mov  r14, rax
L2:  cmp  r13, rbp
     jle  L1
     ...
L3:  call transferToInterpreter
```

Truffle gives you function inlining for free!

Objects

Objects

- Most dynamic languages have a flexible object model
  - Objects are key-value stores
  - Add new properties
  - Change the type of properties
  - But the detailed semantics vary greatly between languages

- Truffle API provides a high-performance, but still customizable object model
  - Single-object storage for objects with few properties
  - Extension arrays for objects with many properties
  - Type specialization, unboxed storage of primitive types
  - Shapes (hidden classes) describe the location of properties

Object API Classes

- `Layout`: one singleton per language that defines basic properties
- `ObjectType`: one singleton of a language-specific subclass
- `Shape`: a list of properties
  - Immutable: adding or deleting a property yields a new Shape
  - Identical series of property additions and deletions yield the same Shape
  - Shape can be invalidated, i.e., superseded by a new Shape with a better storage layout
- `Property`: mapping from a name to a storage location
- `Location`: immutable typed storage location

- `DynamicObject`: storage of the actual data
  - Many `DynamicObject` instances share the same layout described by a Shape

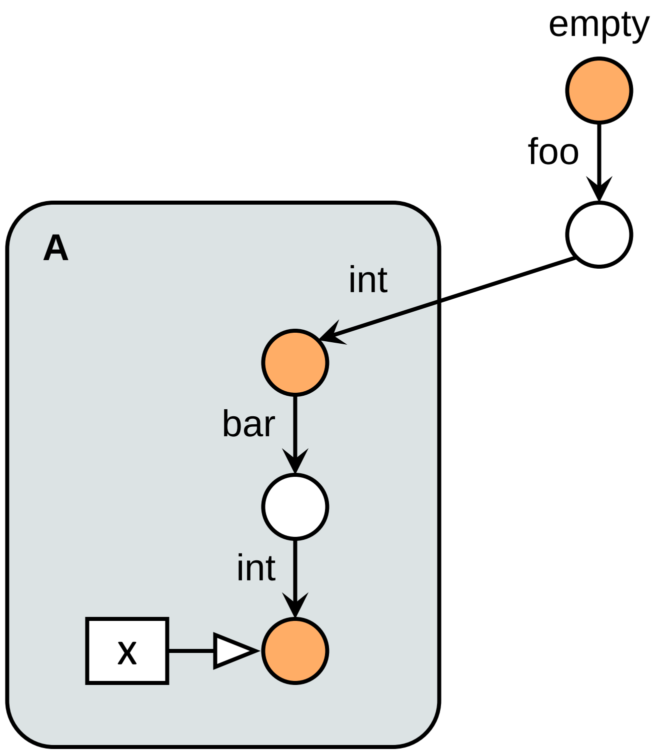
Object Allocation

```
public final class SLContext extends ExecutionContext {
  private static final Layout LAYOUT = Layout.createLayout();

  private final Shape emptyShape = LAYOUT.createShape(SLObjectType.SINGLETON);

  public DynamicObject createObject() {
    return emptyShape.newInstance();
  }

  public static boolean isSLObject(TruffleObject value) {
    return LAYOUT.getType().isInstance(value)
        && LAYOUT.getType().cast(value).getShape().getObjectType() == SLObjectType.SINGLETON;
    }
}
```

```
public final class SLObjectType extends ObjectType {
    public static final ObjectType SINGLETON = new SLObjectType();
}
```

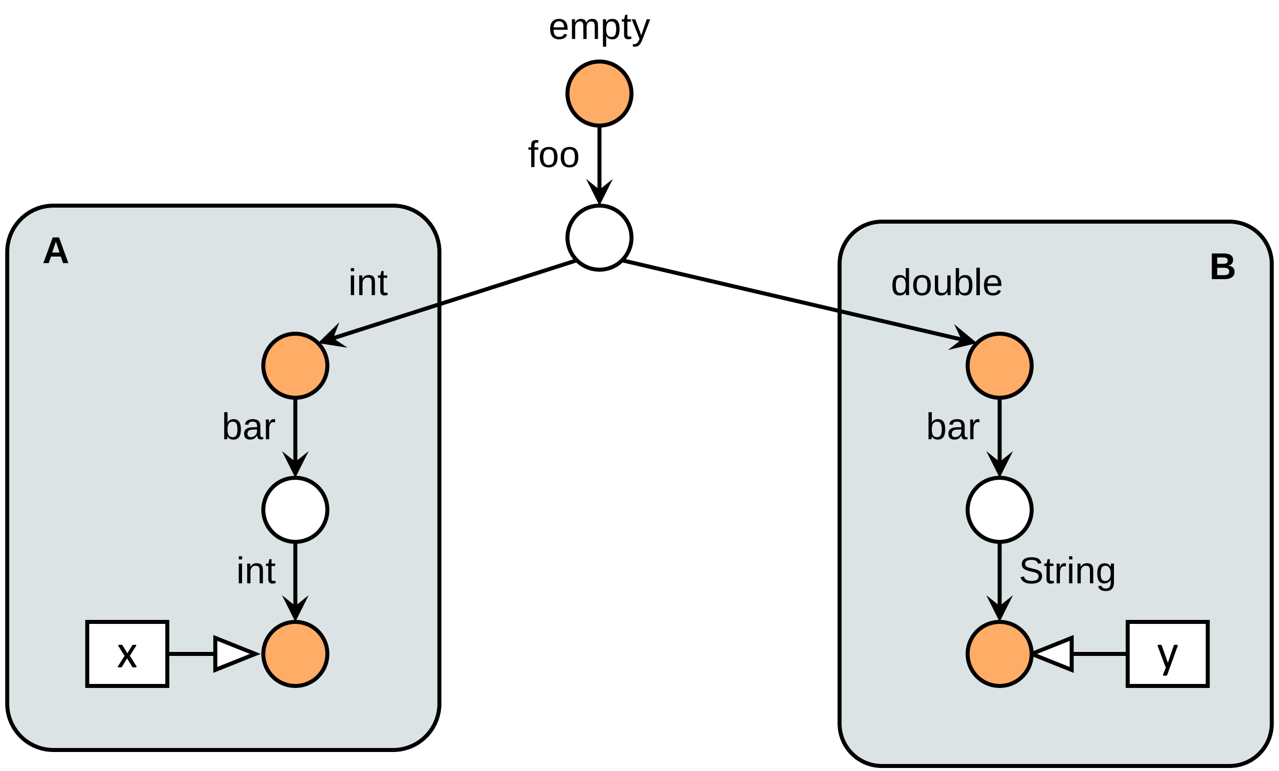
Object Layout Transitions (1)

```
var x = {};
x.foo = 0;
x.bar = 0;
// + subtree A
```



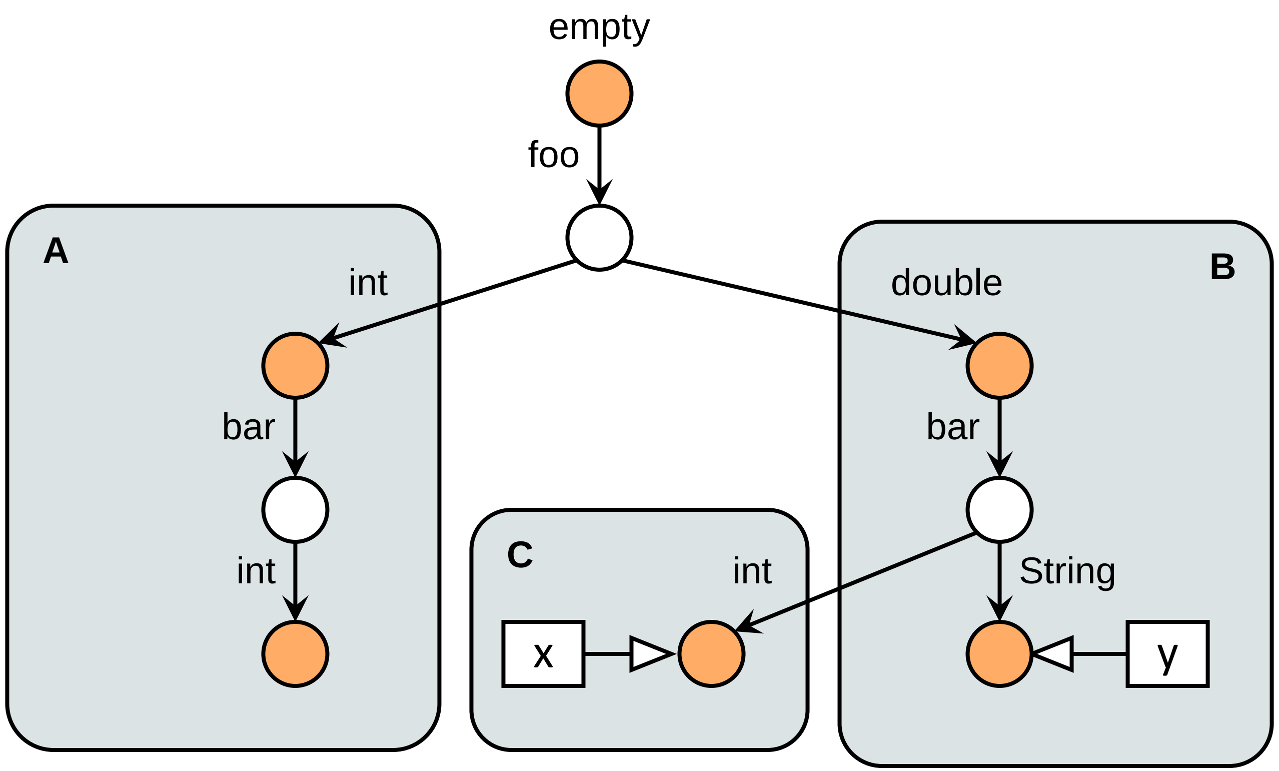
Object Layout Transitions (2)

```
var x = {};
x.foo = 0;
x.bar = 0;
// + subtree A

var y = {};
y.foo = 0.5;
y.bar = "foo";
// + subtree B
```



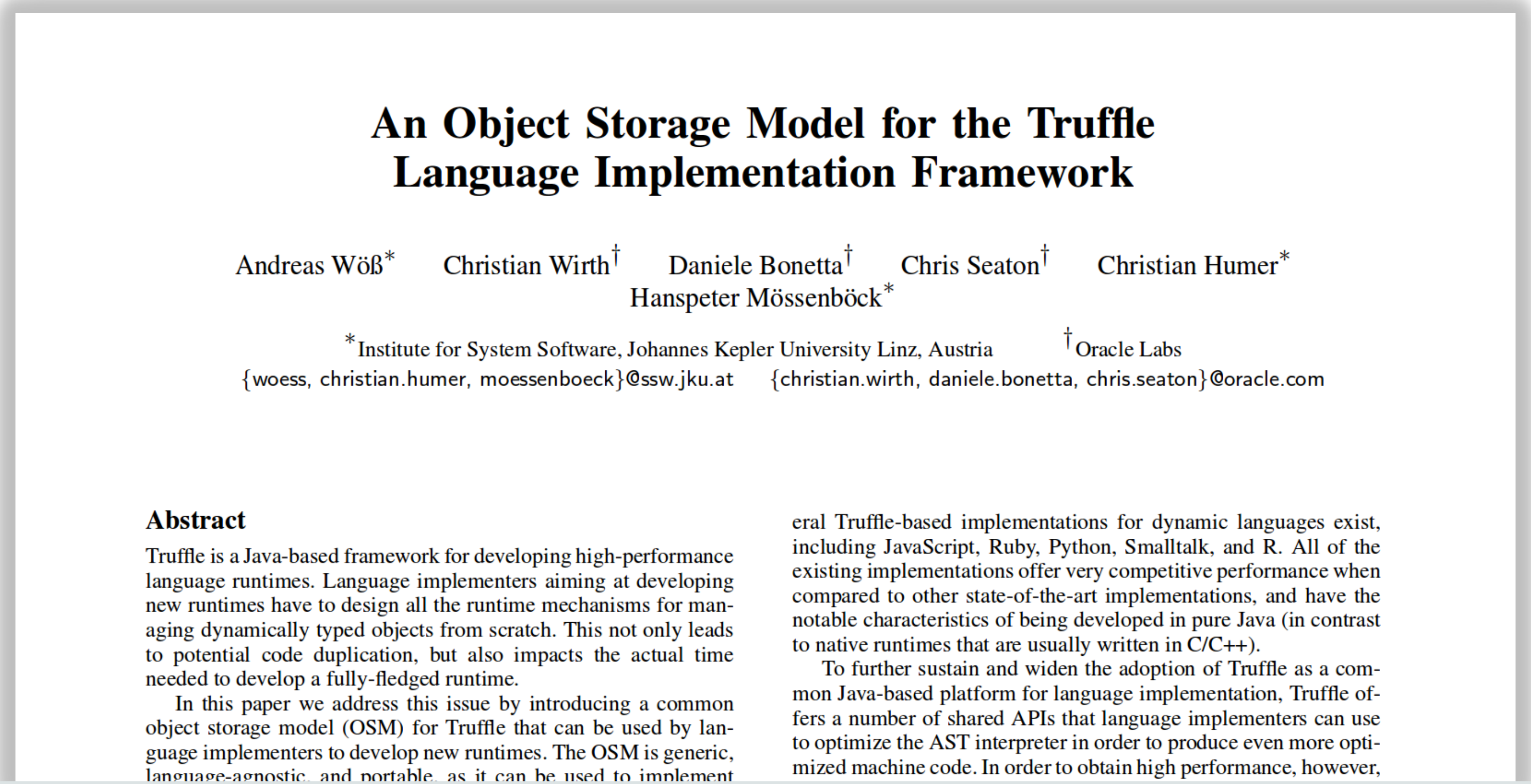
Object Layout Transitions (3)

```
var x = {};
x.foo = 0;
x.bar = 0;
// + subtree A

var y = {};
y.foo = 0.5;
y.bar = "foo";
// + subtree B

x.foo += 0.2
// + subtree C
```



More Details on Object Layout

http://dx.doi.org/10.1145/2647508.2647517



An Object Storage Model for the Truffle Language Implementation Framework

Andreas Wöß[*] Christian Wirth[†] Daniele Bonetta[†] Chris Seaton[†] Christian Humer[*]
Hanspeter Mössenböck[*]

[*] Institute for System Software, Johannes Kepler University Linz, Austria
{woess, christian.humer, moessenboeck}@ssw.jku.at

[†] Oracle Labs
{christian.wirth, daniele.bonetta, chris.seaton}@oracle.com

Abstract

Truffle is a Java-based framework for developing high-performance language runtimes. Language implementers aiming at developing new runtimes have to design all the runtime mechanisms for managing dynamically typed objects from scratch. This not only leads to potential code duplication, but also impacts the actual time needed to develop a fully-fledged runtime.

In this paper we address this issue by introducing a common object storage model (OSM) for Truffle that can be used by language implementers to develop new runtimes. The OSM is generic, language-agnostic, and portable, as it can be used to implement

eral Truffle-based implementations for dynamic languages exist, including JavaScript, Ruby, Python, Smalltalk, and R. All of the existing implementations offer very competitive performance when compared to other state-of-the-art implementations, and have the notable characteristics of being developed in pure Java (in contrast to native runtimes that are usually written in C/C++).

To further sustain and widen the adoption of Truffle as a common Java-based platform for language implementation, Truffle offers a number of shared APIs that language implementers can use to optimize the AST interpreter in order to produce even more optimized machine code. In order to obtain high performance, however,

Polymorphic Inline Cache in SLReadPropertyCacheNode

```java
@Specialization(limit = "CACHE_LIMIT",
                guards = {"namesEqual(cachedName, name)", "shapeCheck(shape, receiver)"},
                assumptions = {"shape.getValidAssumption()"})
protected static Object readCached(DynamicObject receiver, Object name,
                @Cached("name") Object cachedName,
                @Cached("lookupShape(receiver)") Shape shape,
                @Cached("lookupLocation(shape, name)") Location location) {
    return location.get(receiver, shape);
}

@TruffleBoundary
@Specialization(contains = {"readCached"},
                guards = {"isValidSLObject(receiver)"})
protected static Object readUncached(DynamicObject receiver, Object name) {
  Object result = receiver.get(name);
  if (result == null) {
    throw SLUndefinedNameException.undefinedProperty(name);
  }
  return result;
}
```

```java
@Fallback
protected static Object updateShape(Object r, Object name) {
  CompilerDirectives.transferToInterpreter();
  if (!(r instanceof DynamicObject)) {
    throw SLUndefinedNameException.undefinedProperty(name);
  }
  DynamicObject receiver = (DynamicObject) r;
  receiver.updateShape();
  return readUncached(receiver, name);
}
```

Polymorphic Inline Cache in `SLReadPropertyCacheNode`

- Initialization of the inline cache entry (executed infrequently)
  - Lookup the shape of the object
  - Lookup the property name in the shape
  - Lookup the location of the property
  - Values cached in compilation final fields: name, shape, and location
- Execution of the inline cache entry (executed frequently)
  - Check that the name matches the cached name
  - Lookup the shape of the object and check that it matches the cached shape
  - Use the cached location for the read access
    - Efficient machine code because offset and type are compile time constants
- Uncached lookup (when the inline cache size exceeds the limit)
  - Expensive property lookup for every read access
- Fallback
  - Update the object to a new layout when the shape has been invalidated

Polymorphic Inline Cache for Property Writes

- Two different inline cache cases
  - Write a property that does exist
    - No shape transition necessary
    - Guard checks that the type of the new value is the expected constant type
    - Write the new value to a constant location with a constant type
  - Write a property that does not exist
    - Shape transition necessary
    - Both the old and the new shape are @Cached values
    - Write the new constant shape
    - Write the new value to a constant location with a constant type

- Uncached write and Fallback similar to property read

Compilation with Object Allocation

SL source without allocation:

```
function loop(n) {
  i = 0;
  sum = 0;
  while (i <= n) {
    sum = sum + i;
    i = i + 1;
  }
  return sum;
}
```

Machine code without allocation:

```
     mov  r14, 0
     mov  r13, 0
     jmp  L2
L1:  safepoint
     mov  rax, r13
     add  rax, r14
     jo   L3
     inc  r13
     mov  r14, rax
L2:  cmp  r13, rbp
     jle  L1
     ...
L3:  call transferToInterpreter
```

SL source with allocation:

```
function loop(n) {
  o = new();
  o.i = 0;
  o.sum = 0;
  while (o.i <= n) {
    o.sum = o.sum + o.i;
    o.i = o.i + 1;
  }
  return o.sum;
}
```

Machine code with allocation:

```
     mov  r14, 0
     mov  r13, 0
     jmp  L2
L1:  safepoint
     mov  rax, r13
     add  rax, r14
     jo   L3
     inc  r13
     mov  r14, rax
L2:  cmp  r13, rbp
     jle  L1
     ...
L3:  call transferToInterpreter
```

Truffle gives you escape analysis for free!

Stack Walking and Frame Introspection

Stack Walking Requirements

- Requirements
  - Visit all guest language stack frames
    - Abstract over interpreted and compiled frames
  - Allow access to frames down the stack
    - Read and write access is necessary for some languages
  - No performance overhead
    - No overhead in compiled methods as long as frame access is not used
    - No manual linking of stack frames
    - No heap-based stack frames

- Solution in Truffle
  - Stack walking is performed by Java VM
  - Truffle runtime exposes the Java VM stack walking via clean API
  - Truffle runtime abstracts over interpreted and compiled frames
  - Transfer to interpreter used for write access of frames down the stack

Stack Walking

```
public abstract class SLStackTraceBuiltin extends SLBuiltinNode {

  @TruffleBoundary
  private static String createStackTrace() {
    StringBuilder str = new StringBuilder();

    Truffle.getRuntime().iterateFrames(frameInstance -> {
      dumpFrame(str, frameInstance.getCallTarget(), frameInstance.getFrame(FrameAccess.READ_ONLY, true));
      return null;
    });

    return str.toString();
  }

  private static void dumpFrame(StringBuilder str, CallTarget callTarget, Frame frame) {
    if (str.length() > 0) { str.append("\n"); }

    str.append("Frame: ").append(((RootCallTarget) callTarget).getRootNode().toString());
    FrameDescriptor frameDescriptor = frame.getFrameDescriptor();
    for (FrameSlot s : frameDescriptor.getSlots()) {
      str.append(", ").append(s.getIdentifier()).append("=").append(frame.getValue(s));
    }
  }
}
```

`TruffleRuntime` provides stack walking

`FrameInstance` is a handle to a guest language frame

Stack Frame Access

```java
public interface FrameInstance {

  public static enum FrameAccess {
    NONE,
    READ_ONLY,
    READ_WRITE,
    MATERIALIZE
  }

  Frame getFrame(FrameAccess access, boolean slowPath);

  CallTarget getCallTarget();
}
```

The more access you request, the slower it is:
Write access requires transfer to interpreter

Access to the `Frame` and the `CallTarget` gives you full access to your guest language’s data structures and the AST of the method

ORACLE

Polyglot

Language Registration

```
public final class SLMain {

  public static void main(String[] args) throws IOException {
    System.out.println("== running on " + Truffle.getRuntime().getName());

    PolyglotEngine engine = PolyglotEngine.newBuilder().build();
    Source source = Source.fromFileName(args[0]);
    Value result = engine.eval(source);
  }
}
```

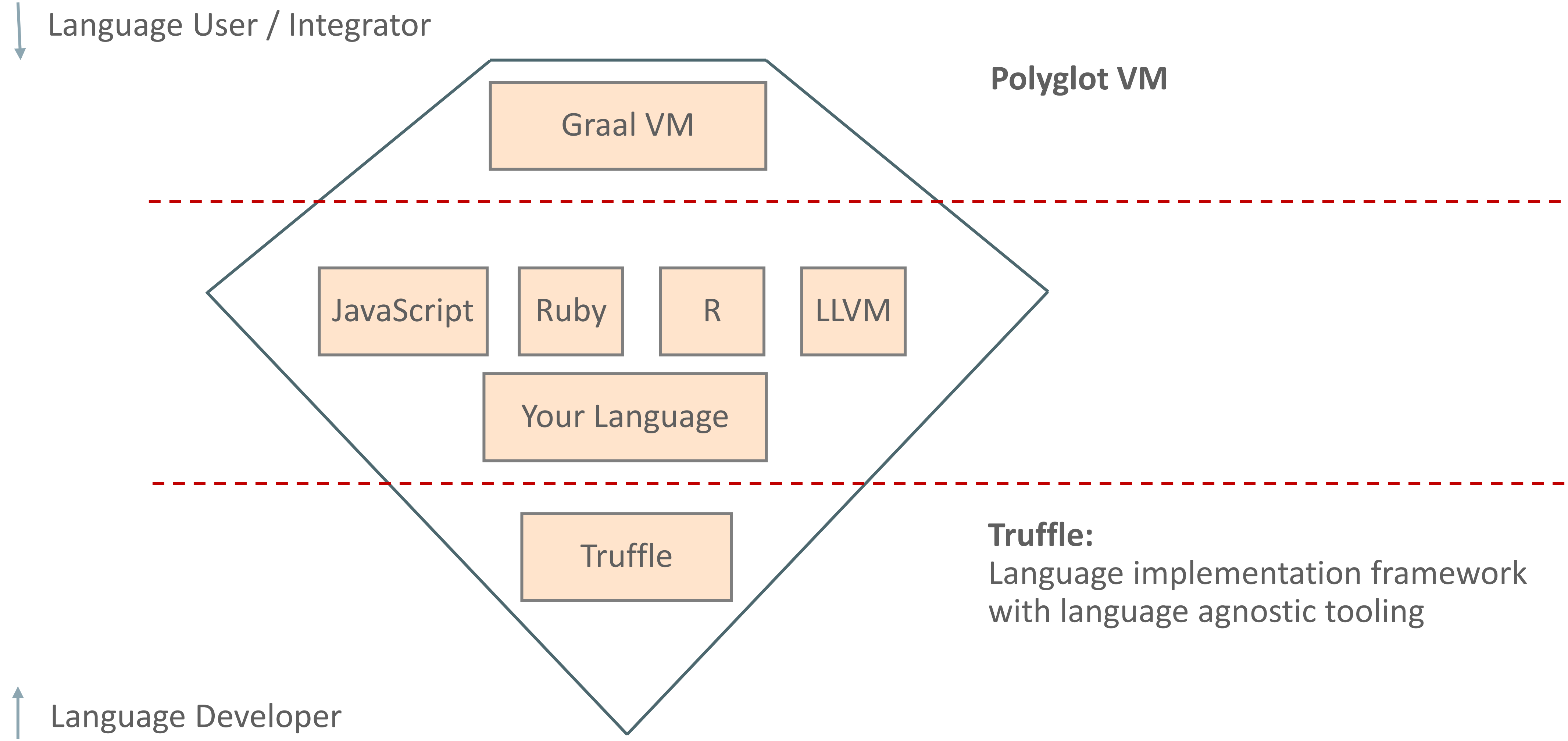
PolyglotEngine is the entry point to execute source code

Language implementation lookup is via mime type

```
@TruffleLanguage.Registration(name = "SL", version = "0.12", mimeType = SLLanguage.MIME_TYPE)
public final class SLLanguage extends TruffleLanguage<SLContext> {

    public static final String MIME_TYPE = "application/x-sl";

    public static final SLLanguage INSTANCE = new SLLanguage();

    @Override
    protected SLContext createContext(Env env) { ... }

    @Override
    protected CallTarget parse(Source source, Node node, String... argumentNames) throws IOException { ... }
```

ORACLE

The Polyglot Diamond



Truffle:
Language implementation framework with language agnostic tooling

Graal VM Multi-Language Shell

Add a vector of numbers using three languages:

```
Ruby>
def rubyadd(a, b)
  a + b;
end
Truffle::Interop.export_method(:rubyadd);

JS>
rubyadd = Interop.import("rubyadd")
function jssum(v) {
  var sum = 0;
  for (var i = 0; i < v.length; i++) {
    sum = Interop.execute(rubyadd, sum, v[i]);
  }
  return sum;
}
Interop.export("jssum", jssum)

R>
v <- runif(1e8);
jssum <- .fastr.interop.import("jssum")
jssum(NULL, v)
```

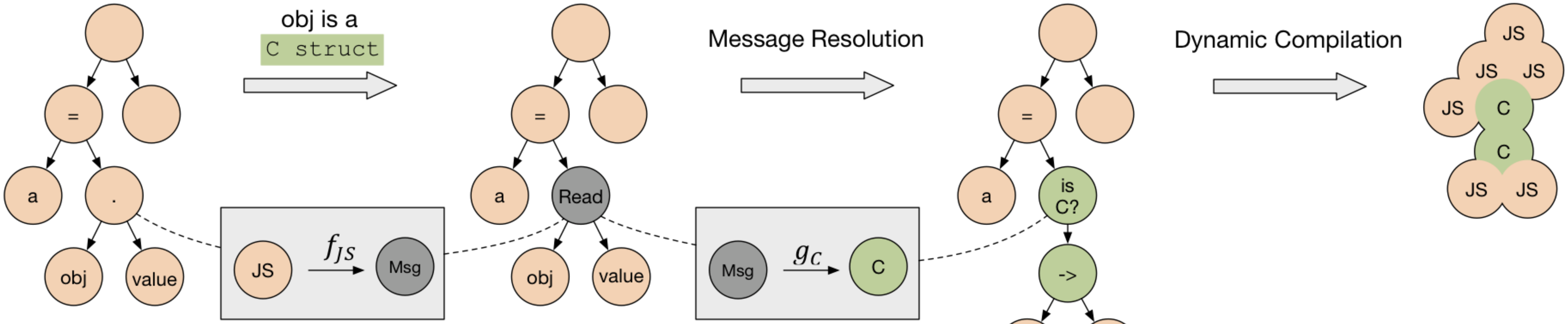
Shell is part of Graal VM download

Start `bin/graalvm`

Explicit export and import of symbols (methods)

High-Performance Language Interoperability (1)

```
var a = obj.value;
```



obj is a `C struct`

Message Resolution

Dynamic Compilation

Map JS access to a message

Map message to a C access

JS-specific object access

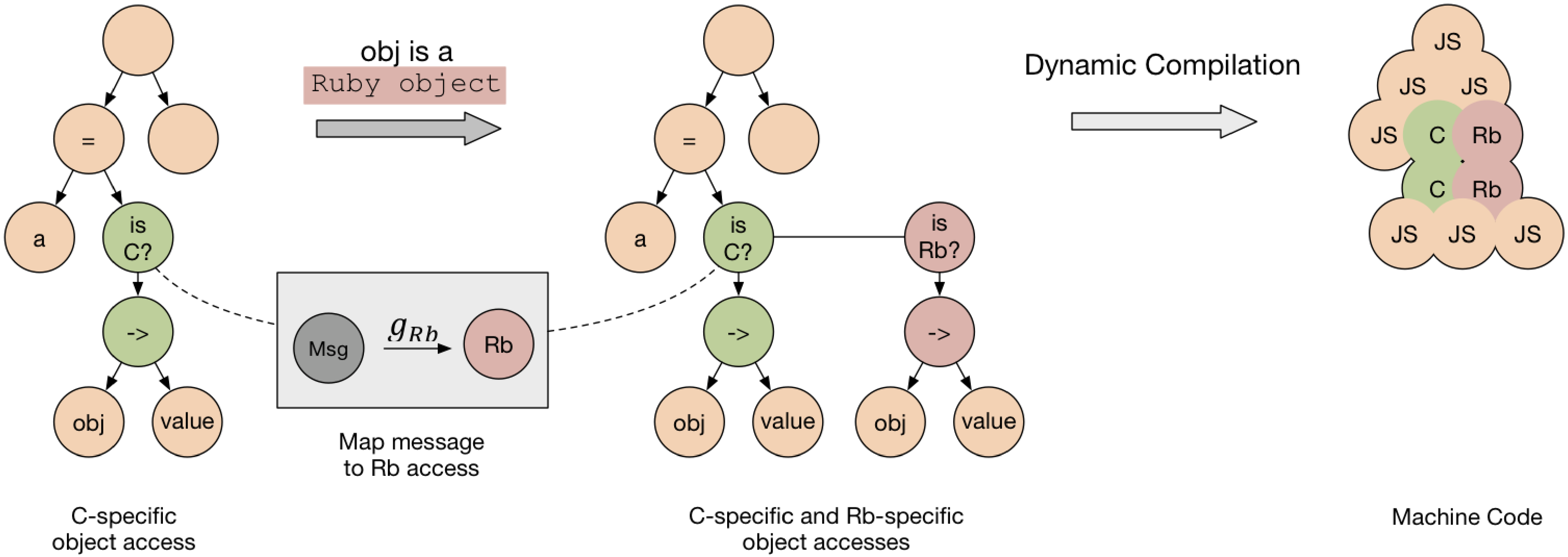
Language-independent object access

C-specific object access

Machine Code

High-Performance Language Interoperability (2)

```
var a = obj.value;
```



Map message to Rb access

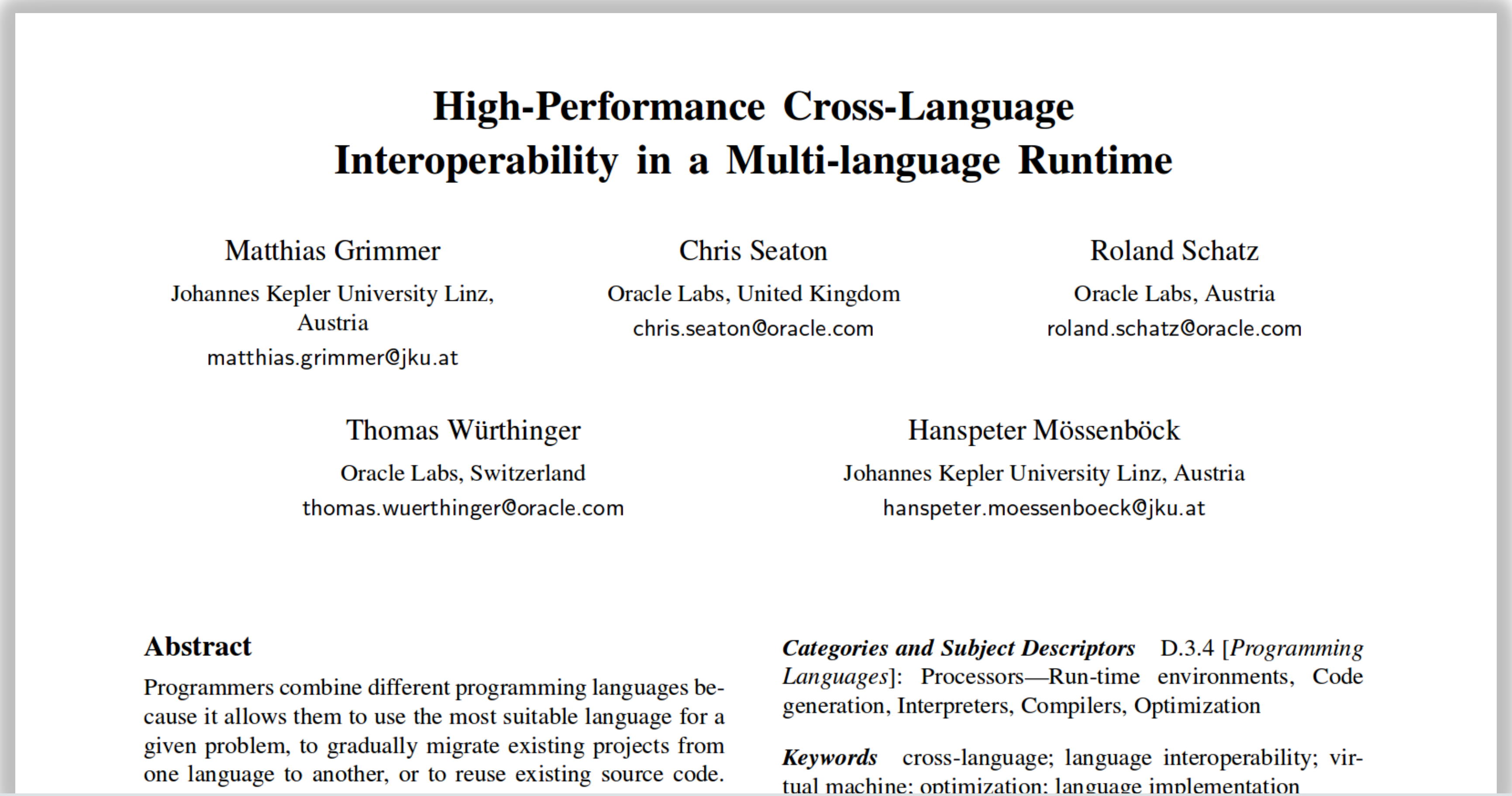
C-specific object access

C-specific and Rb-specific object accesses

Machine Code

More Details on Language Integration

http://dx.doi.org/10.1145/2816707.2816714



High-Performance Cross-Language Interoperability in a Multi-language Runtime

Matthias Grimmer
Johannes Kepler University Linz, Austria
matthias.grimmer@jku.at

Chris Seaton
Oracle Labs, United Kingdom
chris.seaton@oracle.com

Roland Schatz
Oracle Labs, Austria
roland.schatz@oracle.com

Thomas Würthinger
Oracle Labs, Switzerland
thomas.wuerthinger@oracle.com

Hanspeter Mössenböck
Johannes Kepler University Linz, Austria
hanspeter.moessenboeck@jku.at

Abstract

Programmers combine different programming languages because it allows them to use the most suitable language for a given problem, to gradually migrate existing projects from one language to another, or to reuse existing source code.

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

Keywords cross-language; language interoperability; virtual machine; optimization; language implementation

Cross-Language Method Dispatch

```
public abstract class SLDispatchNode extends Node {

  @Specialization(guards = "isForeignFunction(function)")
  protected static Object doForeign(VirtualFrame frame, TruffleObject function, Object[] arguments,
                  @Cached("createCrossLanguageCallNode(arguments)") Node crossLanguageCallNode,
                  @Cached("createToSLTypeNode()") SLForeignToSLTypeNode toSLTypeNode) {
    try {
      Object res = ForeignAccess.sendExecute(crossLanguageCallNode, frame, function, arguments);
      return toSLTypeNode.executeConvert(frame, res);
    } catch (ArityException | UnsupportedTypeException | UnsupportedMessageException e) {
      throw SLUndefinedNameException.undefinedFunction(function);
    }
  }

  protected static boolean isForeignFunction(TruffleObject function) {
      return !(function instanceof SLFunction);
  }
  protected static Node createCrossLanguageCallNode(Object[] arguments) {
    return Message.createExecute(arguments.length).createNode();
  }
  protected static SLForeignToSLTypeNode createToSLTypeNode() {
    return SLForeignToSLTypeNodeGen.create();
  }
}
```

Compilation Across Language Boundaries

Mixed SL and Ruby source code:

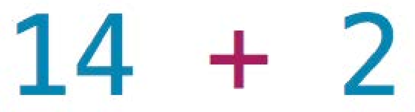
```
function main() {
  eval("application/x-ruby",
       "def add(a, b) a + b; end;");
  eval("application/x-ruby",
       "Truffle::Interop.export_method(:add);");
  ...
}

function loop(n) {
  add = import("add");

  i = 0;
  sum = 0;
  while (i <= n) {
    sum = add(sum, i);
    i = add(i, 1);
  }
  return sum;
}
```

Machine code for loop:

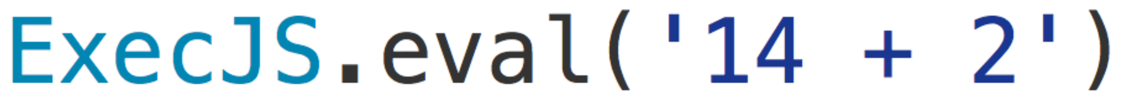
```
     mov  r14, 0
     mov  r13, 0
     jmp  L2
L1:  safepoint
     mov  rax, r13
     add  rax, r14
     jo   L3
     inc  r13
     mov  r14, rax
L2:  cmp  r13, rbp
     jle  L1
     ...
L3:  call transferToInterpreter
```

Truffle gives you language interop for free!

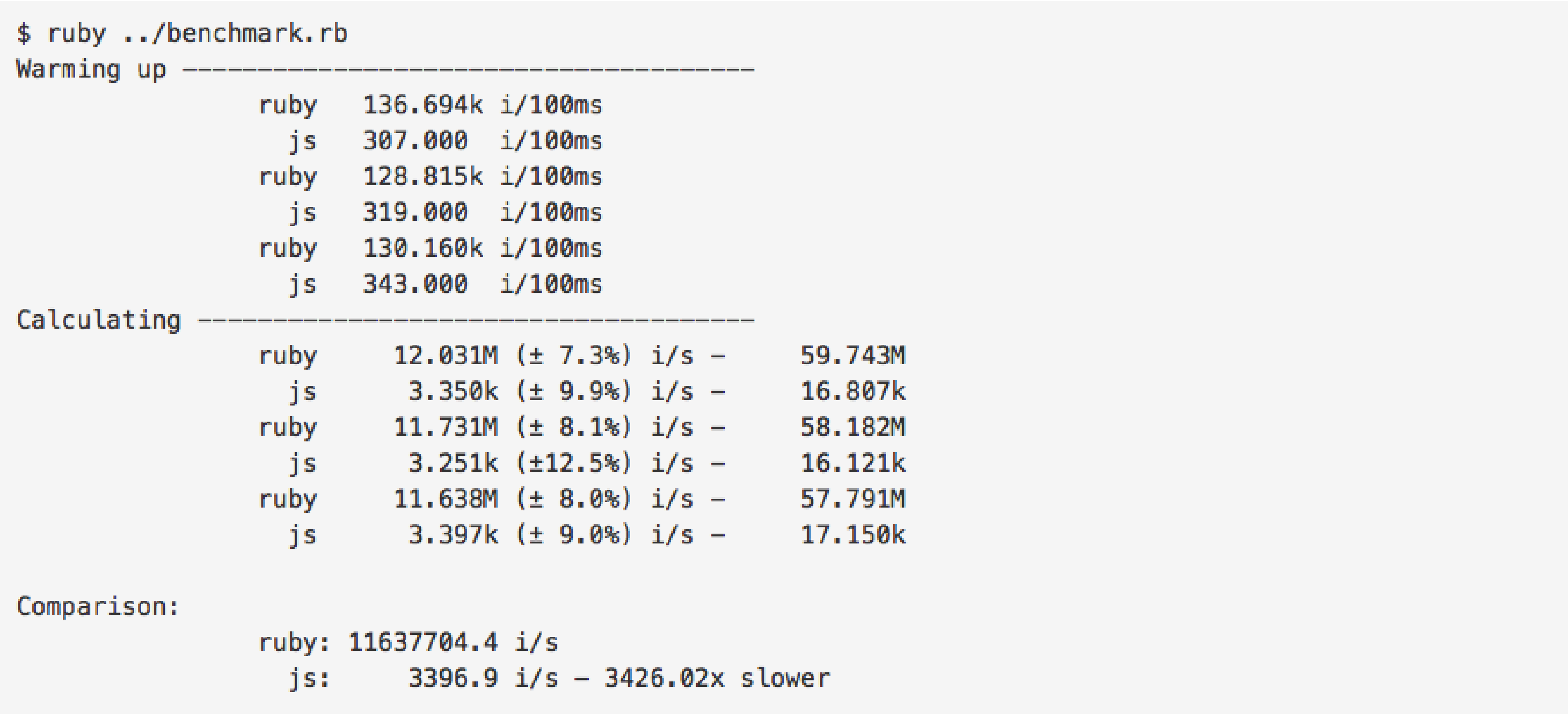
Polyglot Example: Mixing Ruby and JavaScript



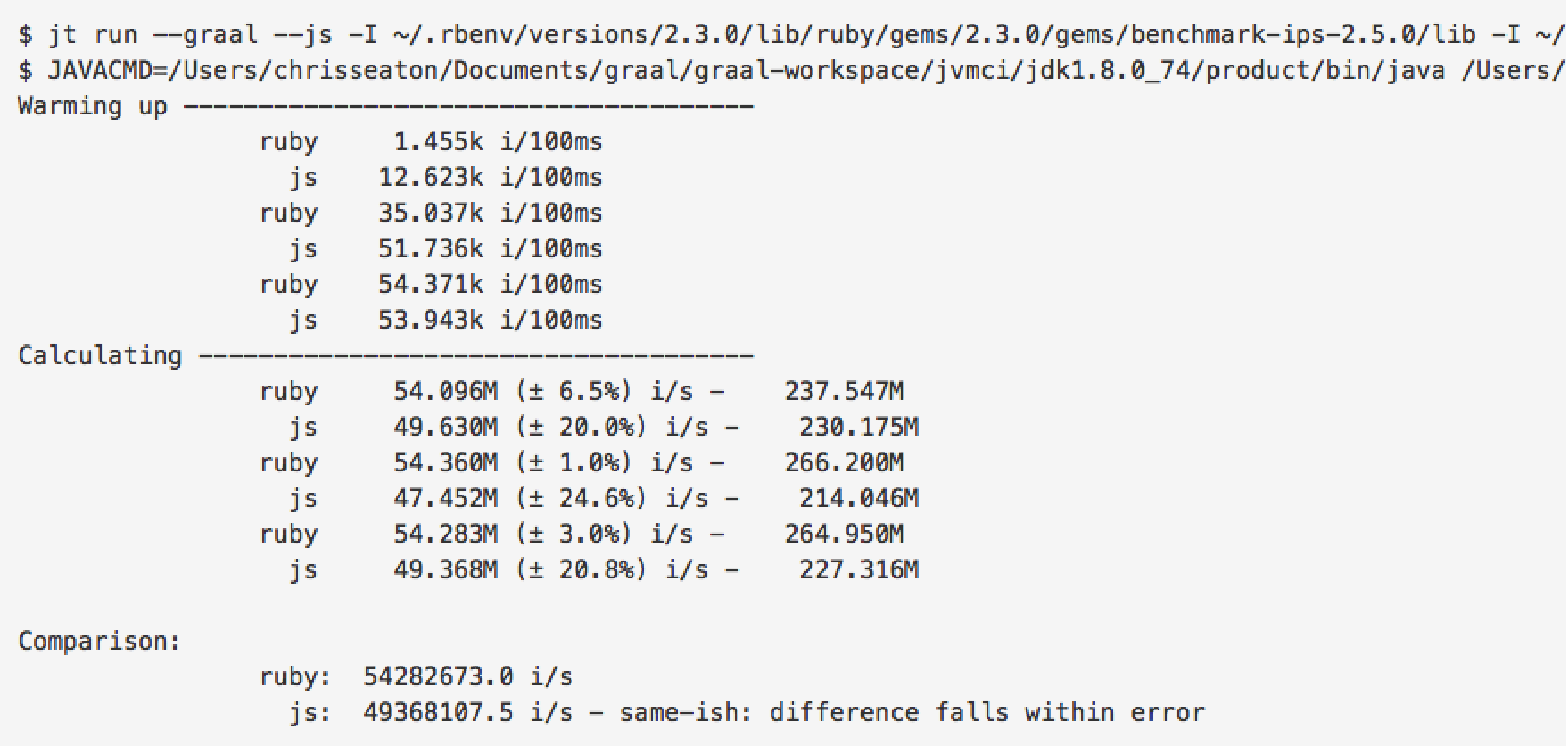
```
14 + 2
```



```
ExecJS.eval('14 + 2')
```



```
$ ruby ../benchmark.rb
Warming up --------------------------------------
                ruby   136.694k i/100ms
                  js   307.000  i/100ms
                ruby   128.815k i/100ms
                  js   319.000  i/100ms
                ruby   130.160k i/100ms
                  js   343.000  i/100ms
Calculating -------------------------------------
                ruby     12.031M (± 7.3%) i/s -     59.743M
                  js      3.350k (± 9.9%) i/s -     16.807k
                ruby     11.731M (± 8.1%) i/s -     58.182M
                  js      3.251k (±12.5%) i/s -     16.121k
                ruby     11.638M (± 8.0%) i/s -     57.791M
                  js      3.397k (± 9.0%) i/s -     17.150k

Comparison:
                ruby: 11637704.4 i/s
                  js:     3396.9 i/s - 3426.02x slower
```



```
$ jt run --graal --js -I ~/.rbenv/versions/2.3.0/lib/ruby/gems/2.3.0/gems/benchmark-ips-2.5.0/lib -I ~/
$ JAVACMD=/Users/chrisseaton/Documents/graal/graal-workspace/jvmci/jdk1.8.0_74/product/bin/java /Users/
Warming up --------------------------------------
                ruby     1.455k i/100ms
                  js    12.623k i/100ms
                ruby    35.037k i/100ms
                  js    51.736k i/100ms
                ruby    54.371k i/100ms
                  js    53.943k i/100ms
Calculating -------------------------------------
                ruby     54.096M (± 6.5%) i/s -    237.547M
                  js     49.630M (± 20.0%) i/s -    230.175M
                ruby     54.360M (± 1.0%) i/s -    266.200M
                  js     47.452M (± 24.6%) i/s -    214.046M
                ruby     54.283M (± 3.0%) i/s -    264.950M
                  js     49.368M (± 20.8%) i/s -    227.316M

Comparison:
                ruby:  54282673.0 i/s
                  js:  49368107.5 i/s - same-ish: difference falls within error
```

Tools

Tools: We Don't Have It All

(Especially for Debuggers)

- Difficult to build
  - Platform specific
  - Violate system abstractions
  - Limited access to execution state

- Productivity tradeoffs for programmers
  - Performance – disabled optimizations
  - Functionality – inhibited language features
  - Complexity – language implementation requirements
  - Inconvenience – nonstandard context (debug flags)

Tools: We Can Have It All

- Build tool support into the Truffle API
  - High-performance implementation
  - Many languages: any Truffle language can be tool-ready with minimal effort
  - Reduced implementation effort

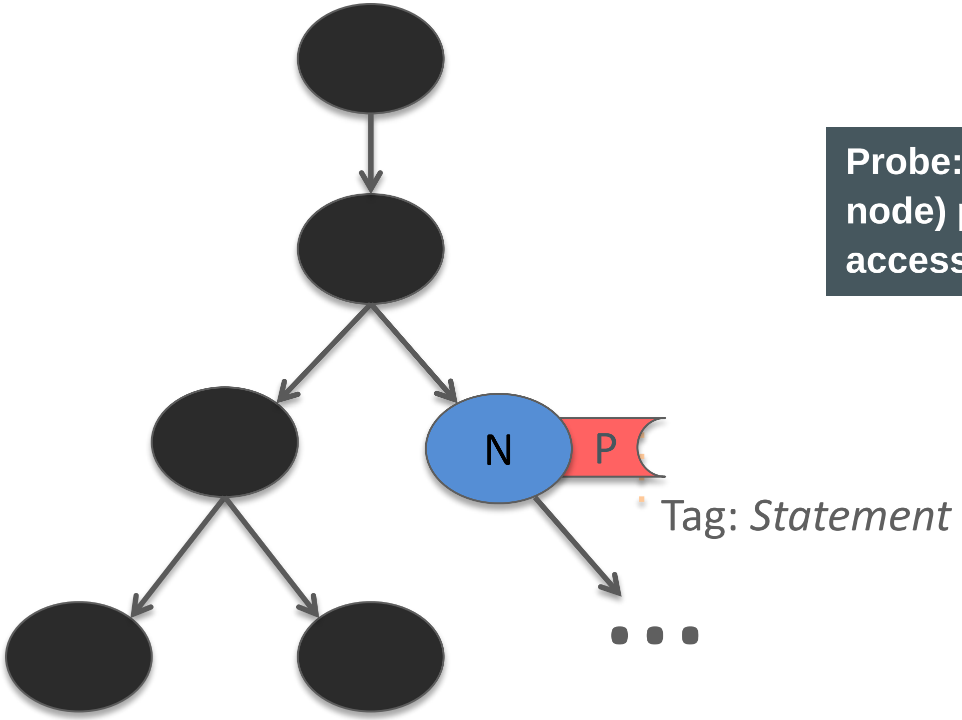
- Generalized *instrumentation* support
  1. Access to execution state & events
  2. Minimal runtime overhead
  3. Reduced implementation effort (for languages *and* tools)

Implementation Effort: Language Implementors

- Treat AST syntax nodes specially
  - Precise source attribution
  - Enable probing
  - Ensure stability
- Add default tags, e.g., Statement, Call, ...
  - Sufficient for many tools
  - Can be extended, adjusted, or replaced dynamically by other tools
- Implement debugging support methods, e.g.
  - Eval a string in context of any stack frame
  - Display language-specific values, method names, …
- More to be added to support new tools & services

ORACLE

“Mark Up” Important AST Nodes for Instrumentation



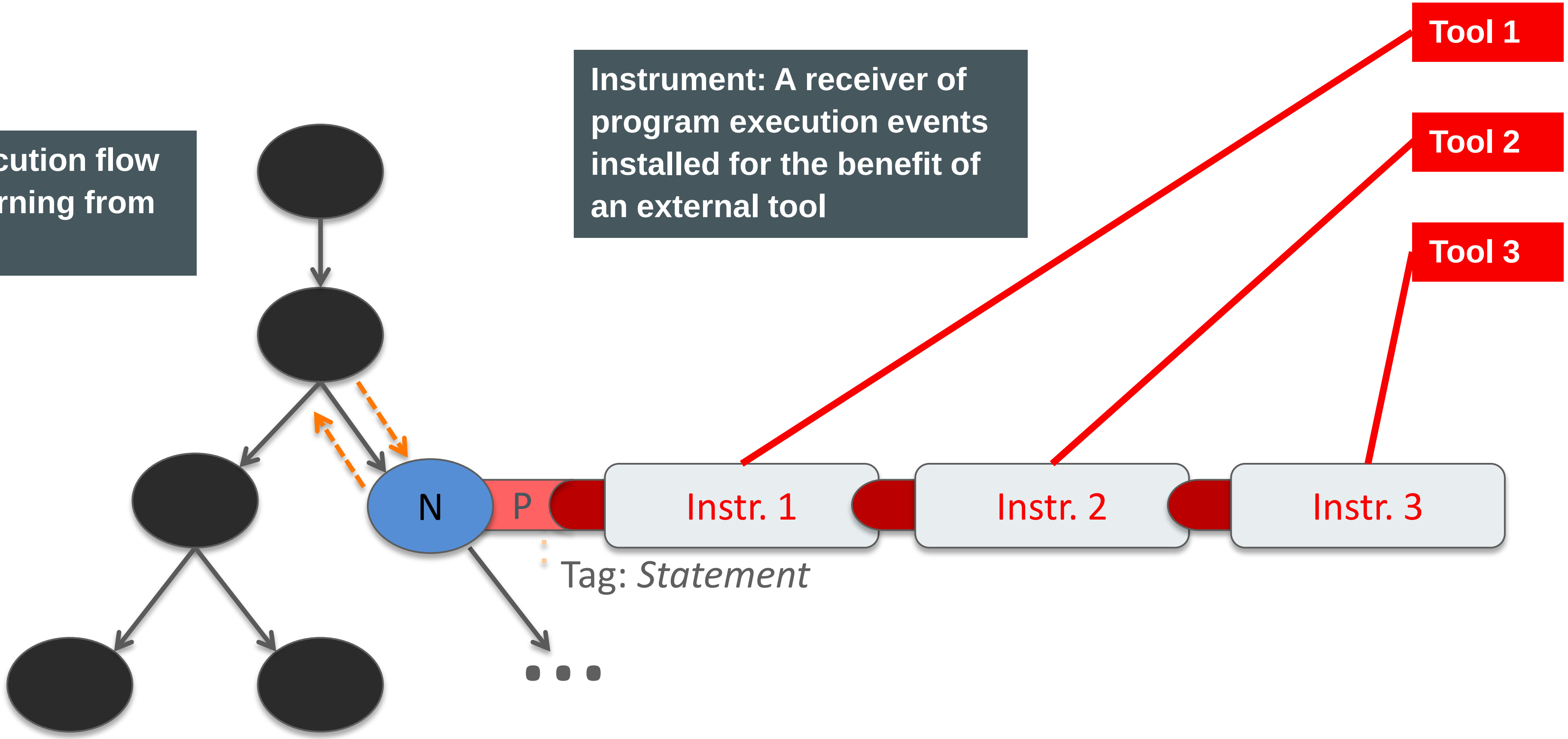
Probe: A program location (AST node) prepared to give tools access to execution state.

Tag: An annotation for configuring tool behavior at a Probe. Multiple tags, possibly tool-specific, are allowed.

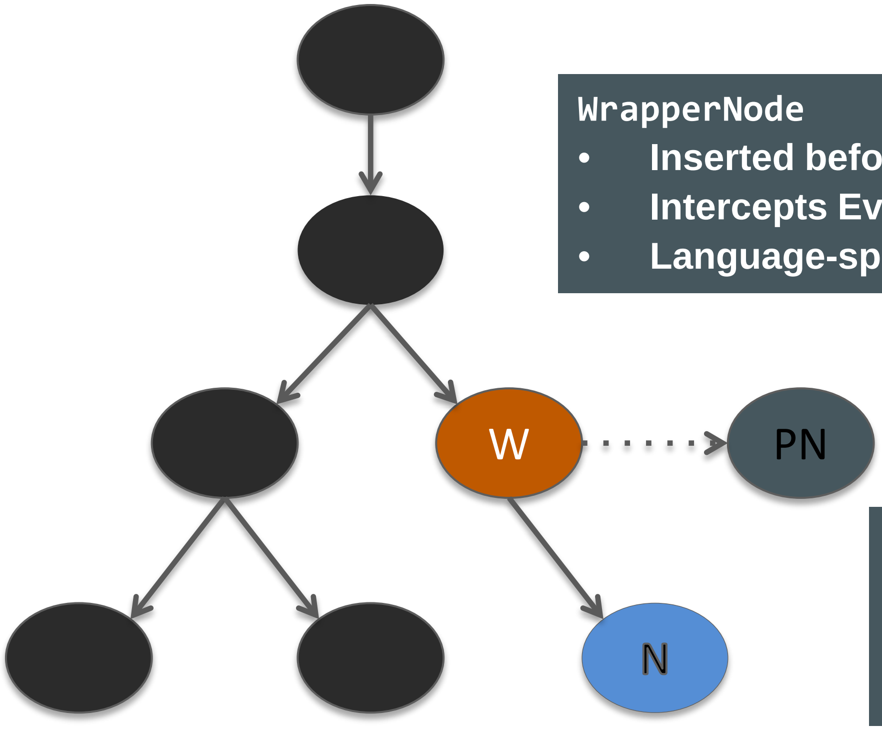
Access to Execution Events

Event: AST execution flow entering or returning from a node.

Instrument: A receiver of program execution events installed for the benefit of an external tool



Implementation: Nodes



`WrapperNode`

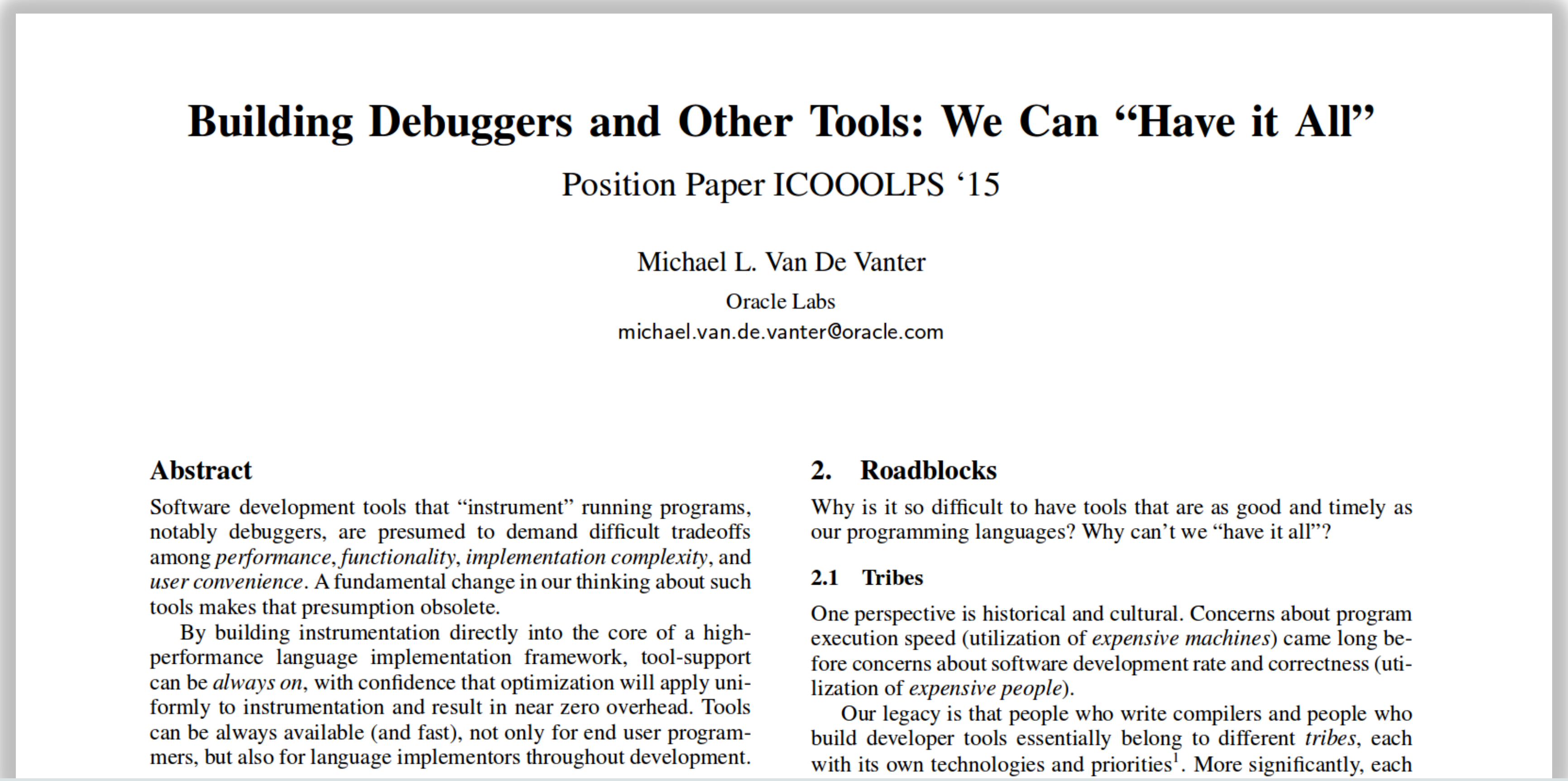
- Inserted before any execution
- Intercepts Events
- Language-specific Type

`ProbeNode`

- Manages “instrument chain” dynamically
- Propagates Events
- Instrumentation Type

More Details on Instrumentation and Debugging

http://dx.doi.org/10.1145/2843915.2843917



Building Debuggers and Other Tools: We Can “Have it All”

Position Paper ICOOOLPS ‘15

Michael L. Van De Vanter

Oracle Labs

michael.van.de.vanter@oracle.com

Abstract

Software development tools that “instrument” running programs, notably debuggers, are presumed to demand difficult tradeoffs among *performance*, *functionality*, *implementation complexity*, and *user convenience*. A fundamental change in our thinking about such tools makes that presumption obsolete.

By building instrumentation directly into the core of a high-performance language implementation framework, tool-support can be *always on*, with confidence that optimization will apply uniformly to instrumentation and result in near zero overhead. Tools can be always available (and fast), not only for end user programmers, but also for language implementors throughout development.

2. Roadblocks

Why is it so difficult to have tools that are as good and timely as our programming languages? Why can’t we “have it all”?

2.1 Tribes

One perspective is historical and cultural. Concerns about program execution speed (utilization of *expensive machines*) came long before concerns about software development rate and correctness (utilization of *expensive people*).

Our legacy is that people who write compilers and people who build developer tools essentially belong to different *tribes*, each with its own technologies and priorities[1]. More significantly, each

Node Tags

```
@Instrumentable(factory = SLStatementNodeWrapper.class)
public abstract class SLStatementNode extends Node {

  private boolean hasStatementTag;
  private boolean hasRootTag;

  @Override
  protected boolean isTaggedWith(Class<?> tag) {
      if (tag == StandardTags.StatementTag.class) {
          return hasStatementTag;
      } else if (tag == StandardTags.RootTag.class) {
          return hasRootTag;
      }
      return false;
  }
}
```

Annotation generates type-specialized `WrapperNode`

The set of tags is extensible, tools can provide new tags

Example: Debugger

Simple command line debugger is in Truffle development repository: https://github.com/graalvm/truffle#hacking-truffle

```
mx repl
==> GraalVM Polyglot Debugger 0.9
Copyright (c) 2013-6, Oracle and/or its affiliates
    Languages supported (type "lang <name>" to set default)
    JS ver. 0.9
    SL ver. 0.12
()  loads LoopPrint.sl
Frame 0 in LoopPrint.sl
       1  function loop(n) {
       2    i = 0;
       3    while (i < n) {
       4      i = i + 1;
       5    }
       6    return i;
       7  }
       8
       9  function main() {
-->   10    i = 0;
      11    while (i < 20) {
      12      loop(1000);
      13      i = i + 1;
      14    }
      15    println(loop(1000));
      16  }
```

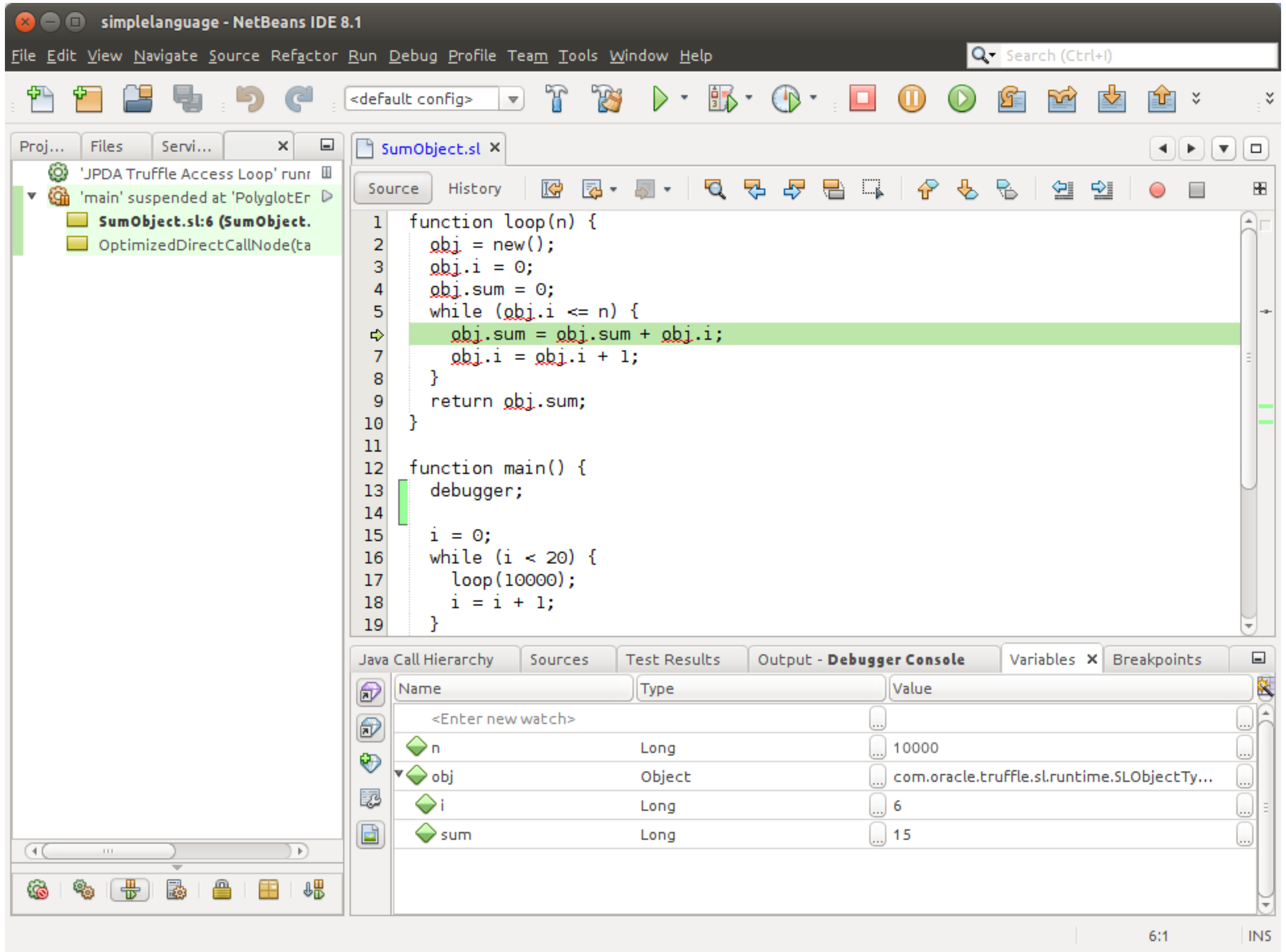
```
(<1> LoopPrint.sl:10)( SL ) break 4
==> breakpoint 0 set at LoopPrint.sl:4
(<1> LoopPrint.sl:10)( SL ) continue
Frame 0 in LoopPrint.sl
[...]
-->   4      i = i + 1;
[...]
(<1> LoopPrint.sl:4)( SL ) frame
==> Frame 0:
    #0: n = 1000
    #1: i = 0
(<1> LoopPrint.sl:4)( SL ) step
Frame 0 in LoopPrint.sl
[...]
-->   3    while (i < n) {
[...]
(<1> LoopPrint.sl:3)( SL ) frame
==> Frame 0:
    #0: n = 1000
    #1: i = 1
(<1> LoopPrint.sl:3)( SL ) backtrace
==>   0: at LoopPrint.sl:3 in root loop    line="  while (i <
      1: at LoopPrint.sl:12~ in root main    line="    loop(1
```

NetBeans Debugger

- NetBeans has experimental Truffle debugging support
- Download latest nightly build of NetBeans
  - This demo uses nightly build 201606100002
- Install Truffle plugin
  - Tools -> Plugins -> Available Plugins -> search for "Truffle"
  - Install "Truffle Debugging Support"

- Start SL in debug mode
  - `sl -debug tests/SumObject.sl`
- Manually insert `debugger;` statement into SumObject.sl
- Attach NetBeans debugger to port 8000

Example: NetBeans Debugger


simplelanguage - NetBeans IDE 8.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
Proj... Files Servi...
'JPDA Truffle Access Loop' runr
'main' suspended at 'PolyglotEr
SumObject.sl:6 (SumObject.
OptimizedDirectCallNode(ta
SumObject.sl
Source History
Java Call Hierarchy Sources Test Results Output - Debugger Console Variables Breakpoints
6:1 INS

```
function loop(n) {
  obj = new();
  obj.i = 0;
  obj.sum = 0;
  while (obj.i <= n) {
    obj.sum = obj.sum + obj.i;
    obj.i = obj.i + 1;
  }
  return obj.sum;
}

function main() {
  debugger;

  i = 0;
  while (i < 20) {
    loop(10000);
    i = i + 1;
  }
```

Name	Type	Value
<Enter new watch>		
n	Long	10000
obj	Object	com.oracle.truffle.sl.runtime.SLObjectTy...
i	Long	6
sum	Long	15

`sl -debug tests/SumObject.sl`

`debugger;` statement sets a breakpoint manually because NetBeans does not know .sl files

Stepping and Variables view work as expected

Stacktrace view has small rendering issues

ORACLE

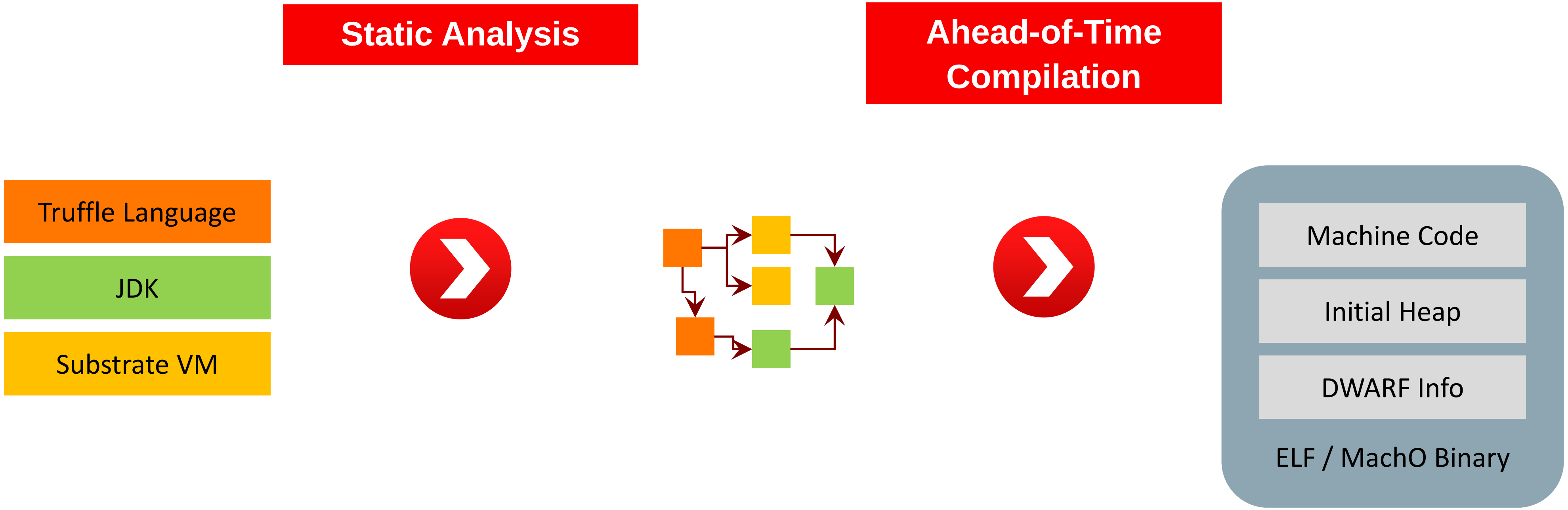
Substrate VM

Substrate VM

- Goal
  - Run Truffle languages without the overhead of a Java VM

- Approach
  - Ahead-of-time compile the Java bytecodes to machine code
  - Build standard Linux / MacOS executable

Substrate VM: Execution Model



Static Analysis

Ahead-of-Time Compilation

Truffle Language

JDK

Substrate VM

Machine Code

Initial Heap

DWARF Info

ELF / MachO Binary

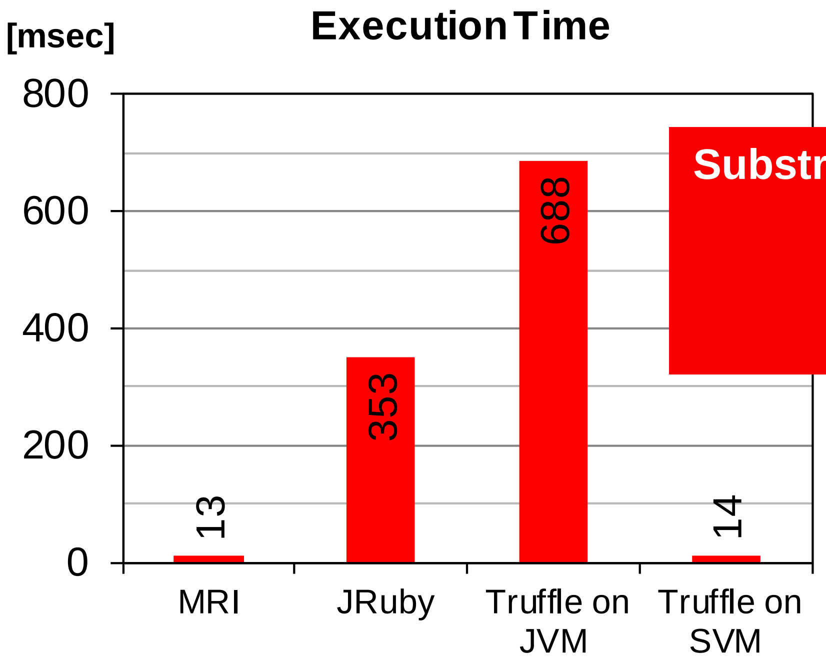
All Java classes from
Truffle language
(or any application),
JDK, and Substrate VM

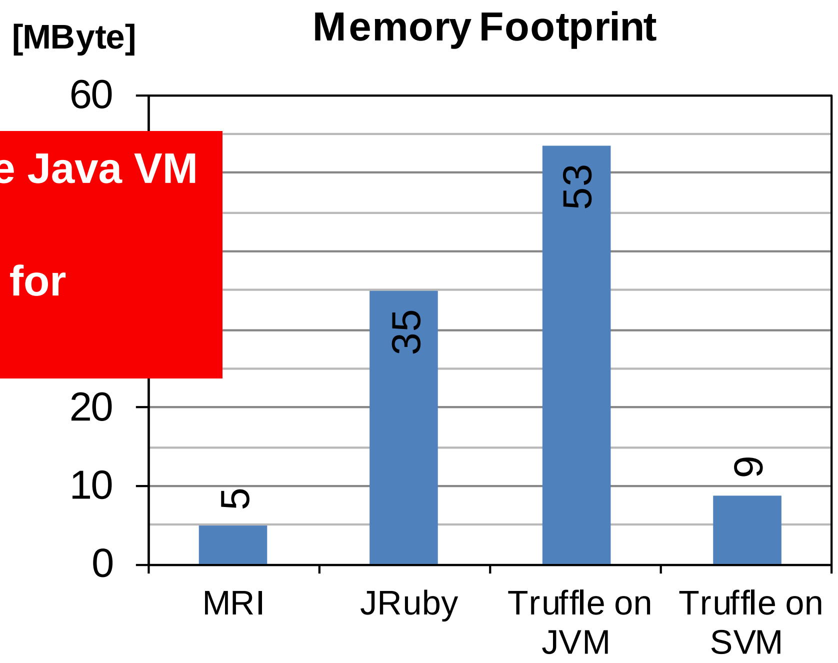
Reachable methods,
fields, and classes

Application running
without dependency on JDK
and without Java class loading

Substrate VM: Startup Performance

Running Ruby “Hello World”





Substrate VM eliminates the Java VM startup overhead: Orders of magnitude for time and memory

Execution time: `time -f "%e"`

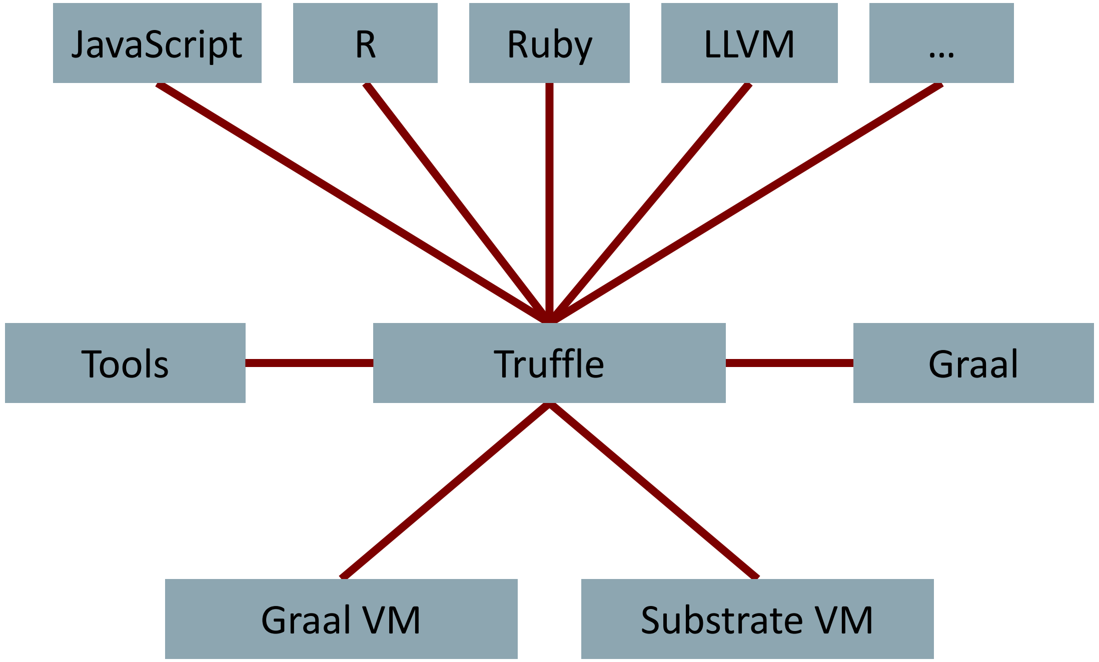
Memory footprint: `time -f "%M"`

Summary

Summary

AST Interpreter for every language

Your language should be here!



Common API separates language implementation, optimization system, and tools (debugger)

Language agnostic dynamic compiler

Integrate with Java applications

Low-footprint VM, also suitable for embedding

Integrated Cloud

Applications & Platform Services

ORACLE®