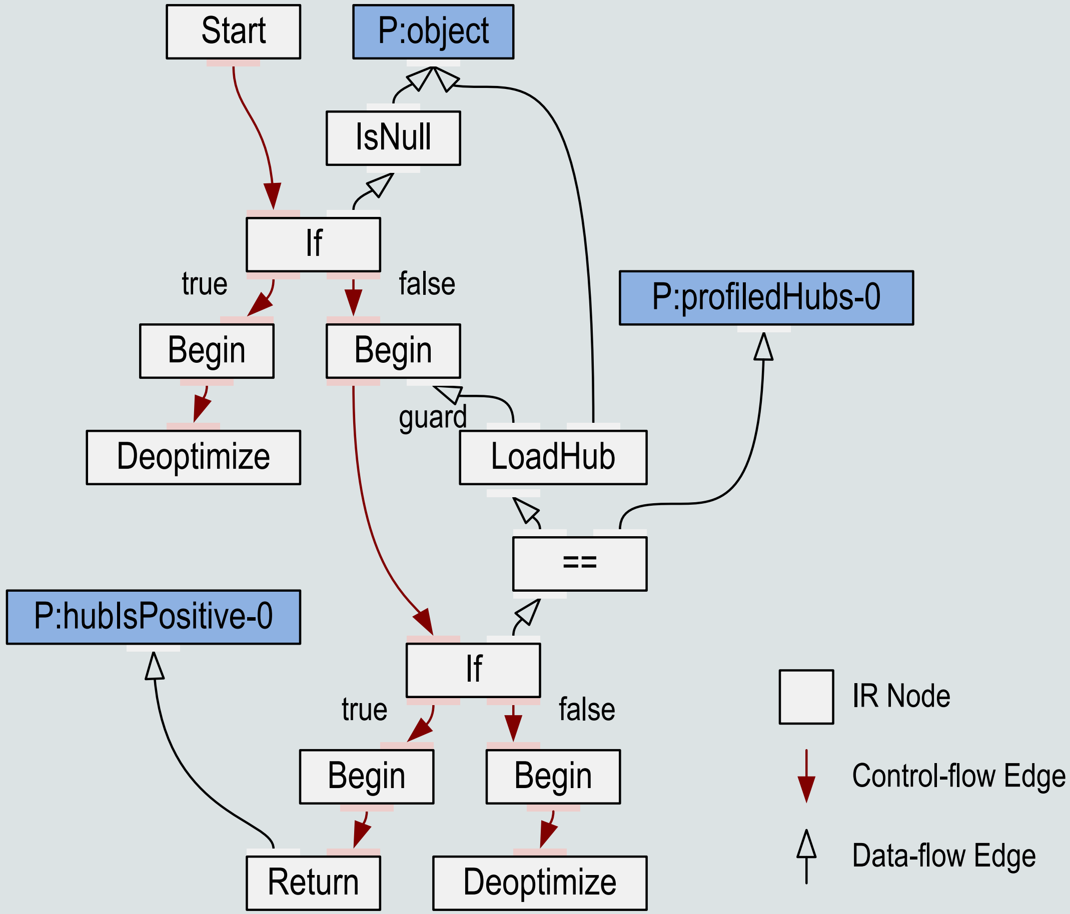
Self-Specialising Interpreters and Partial Evaluation

Graal and Truffle

Chris Seaton
Research Manager
Oracle Labs
9 August 2016



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. Oracle reserves the right to alter its development plans and practices at any time, and 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.

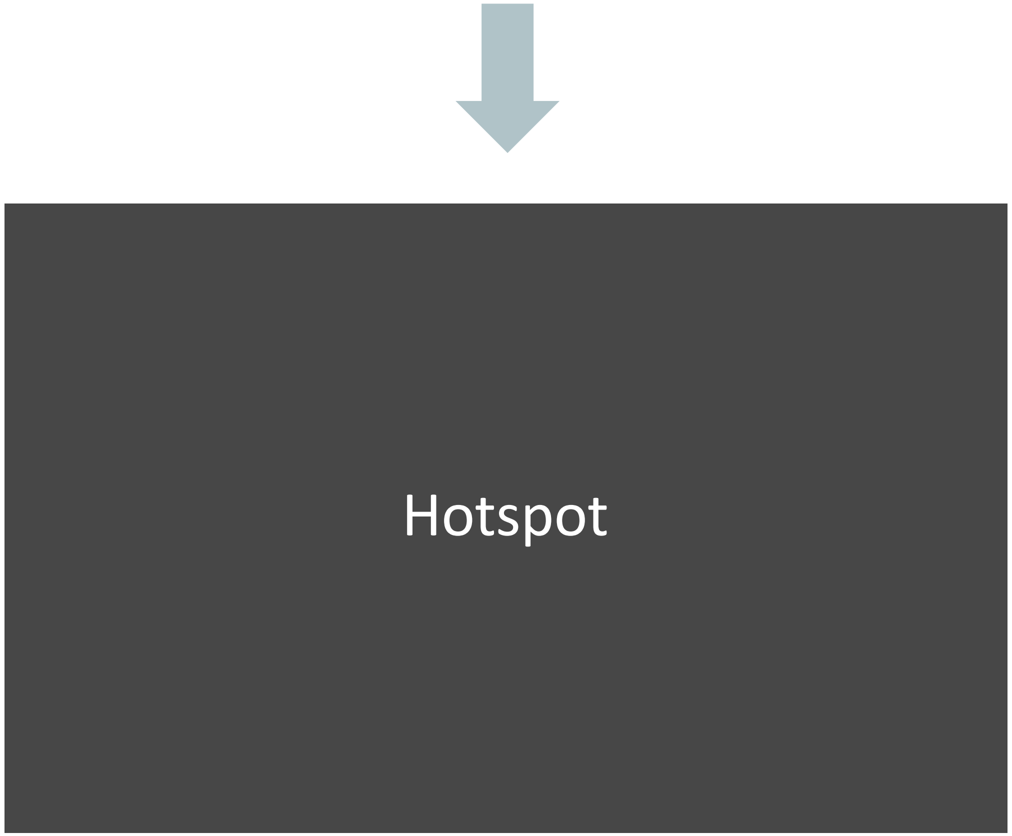
Compilers are, of course, metaprogramming systems

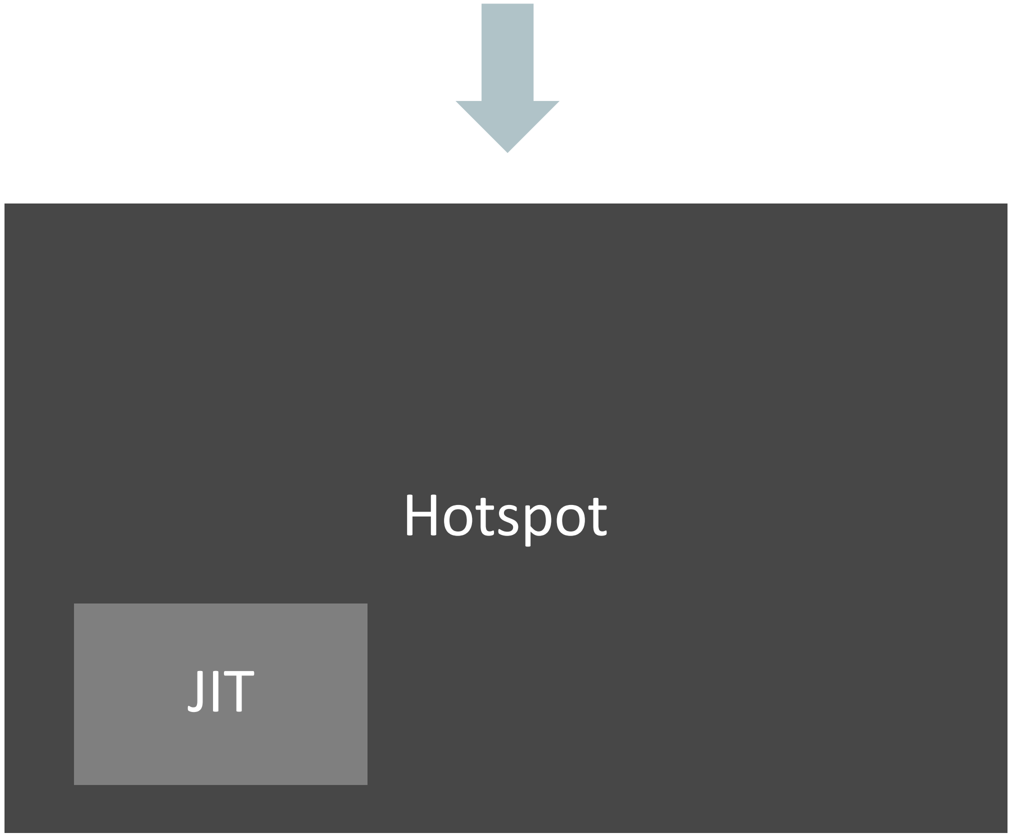
Writing languages that target the JVM

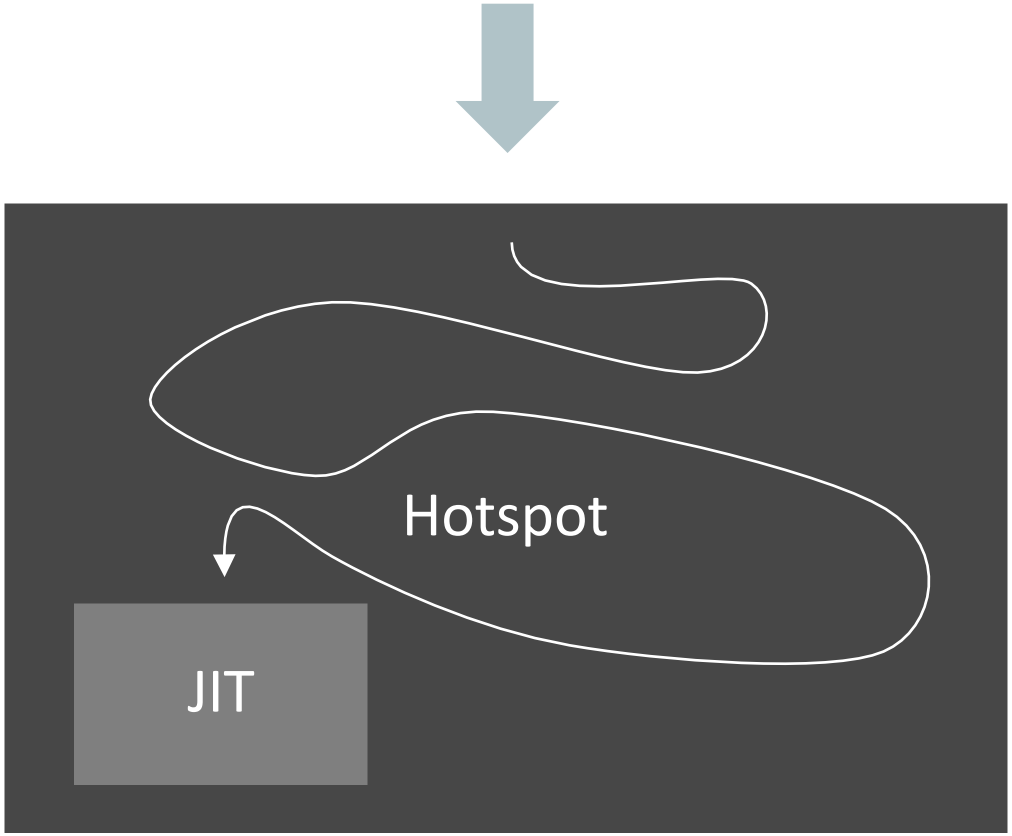
```
0:    iconst_2
1:    istore_1
2:    iload_1
3:    sipush  1000
6:    if_icmpge       44
9:    iconst_2
10:   istore_2
11:   iload_2
12:   iload_1
13:   if_icmpge       31
16:   iload_1
17:   iload_2
18:   irem
19:   ifne    25
22:   goto    38
25:   iinc    2, 1
28:   goto    11
31:   getstatic       #84; // Field java/lang/System.out:Ljava/io/PrintStream;
34:   iload_1
35:   invokevirtual   #85; // Method java/io/PrintStream.println:(I)V
38:   iinc    1, 1
41:   goto    2
44:   return
```

ORACLE

Hotspot


Hotspot


Hotspot
JIT


Hotspot
JIT

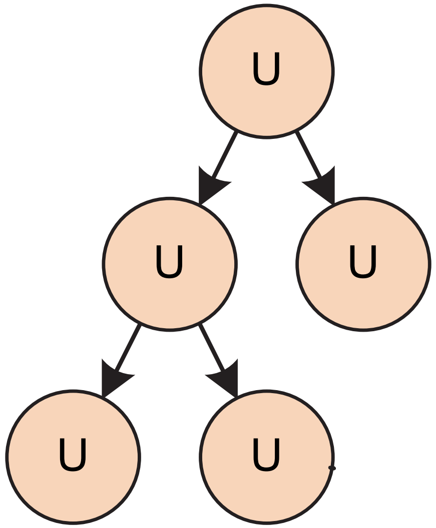

JIT
(Graal)
Hotspot
JIT


Truffle
JIT
(Graal)
Hotspot
JIT

Two levels of program representation

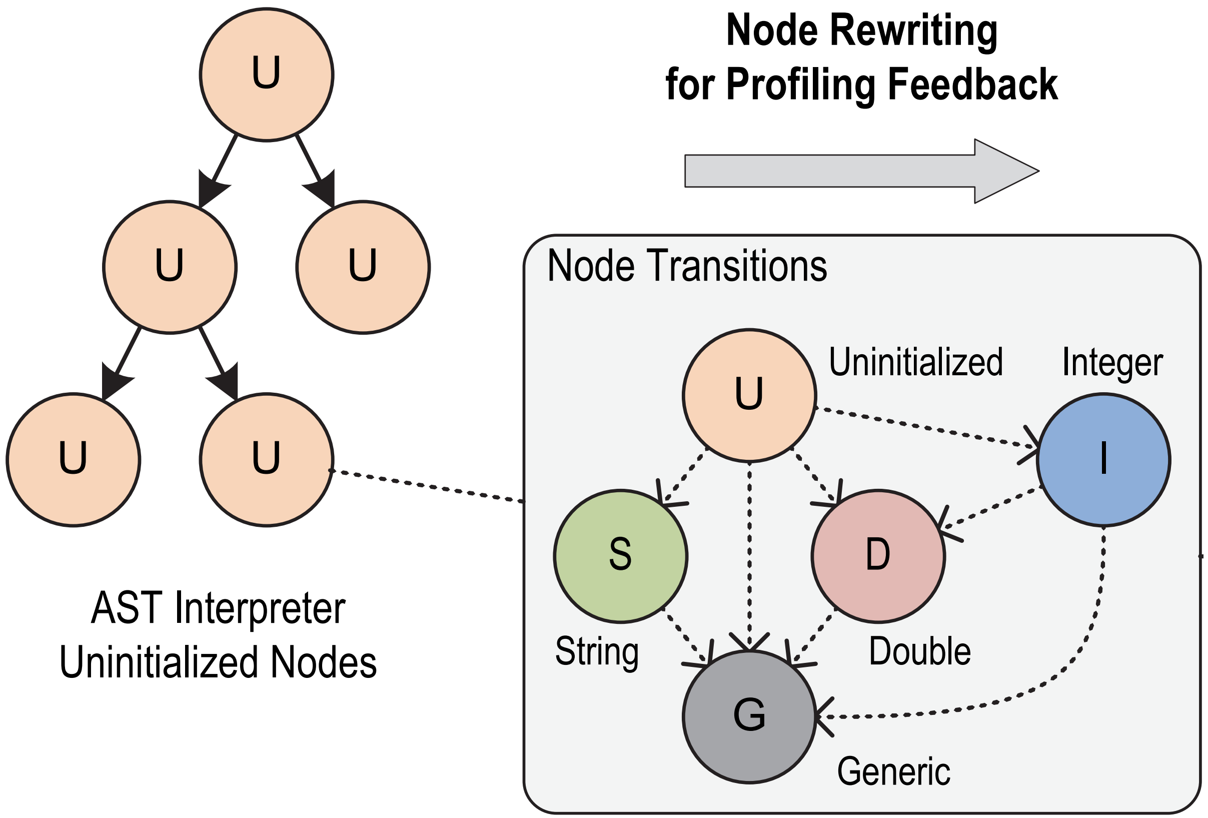
- *Truffle – ASTs*
- *Graal – compiler IR*

Truffle

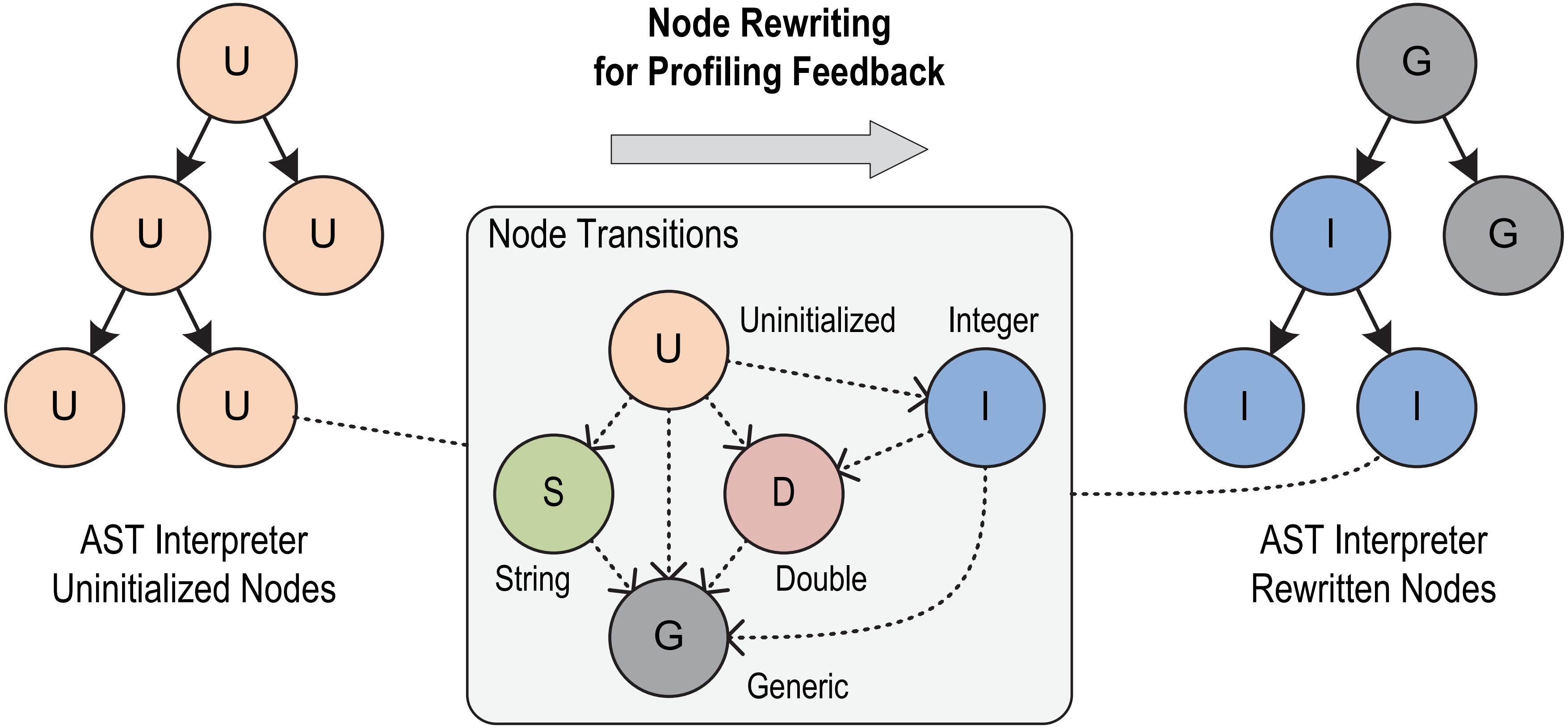


AST Interpreter
Uninitialized Nodes

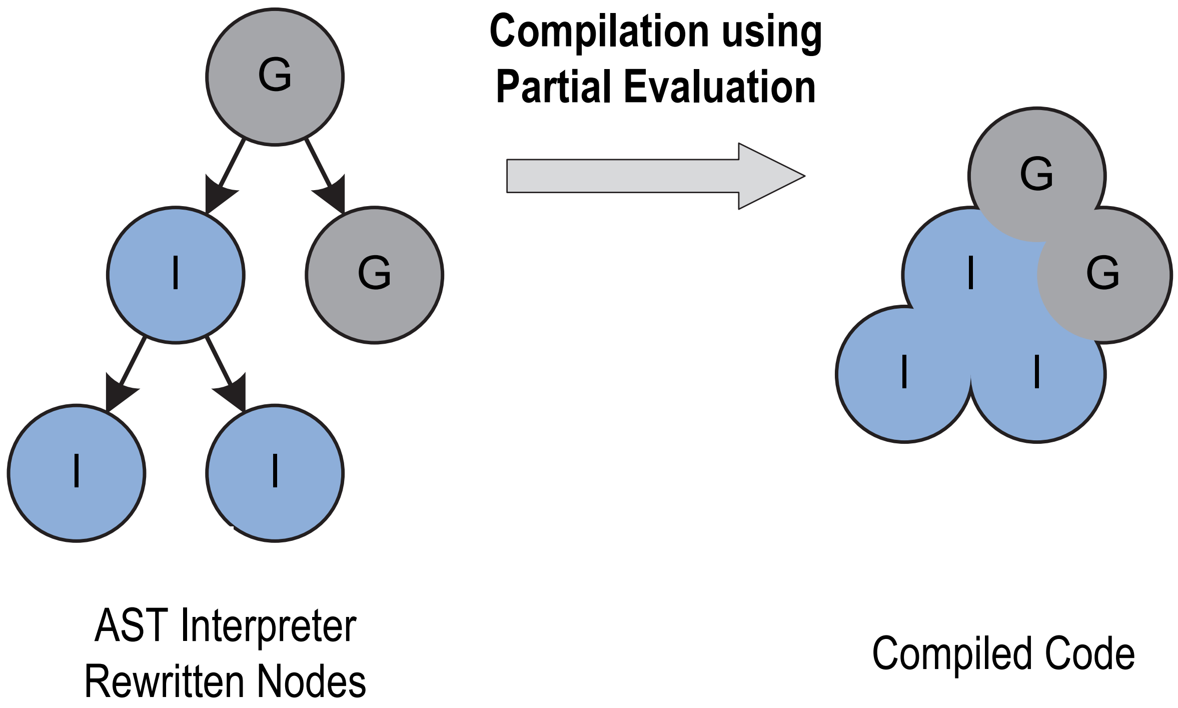
T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In Proceedings of Onward!, 2013.



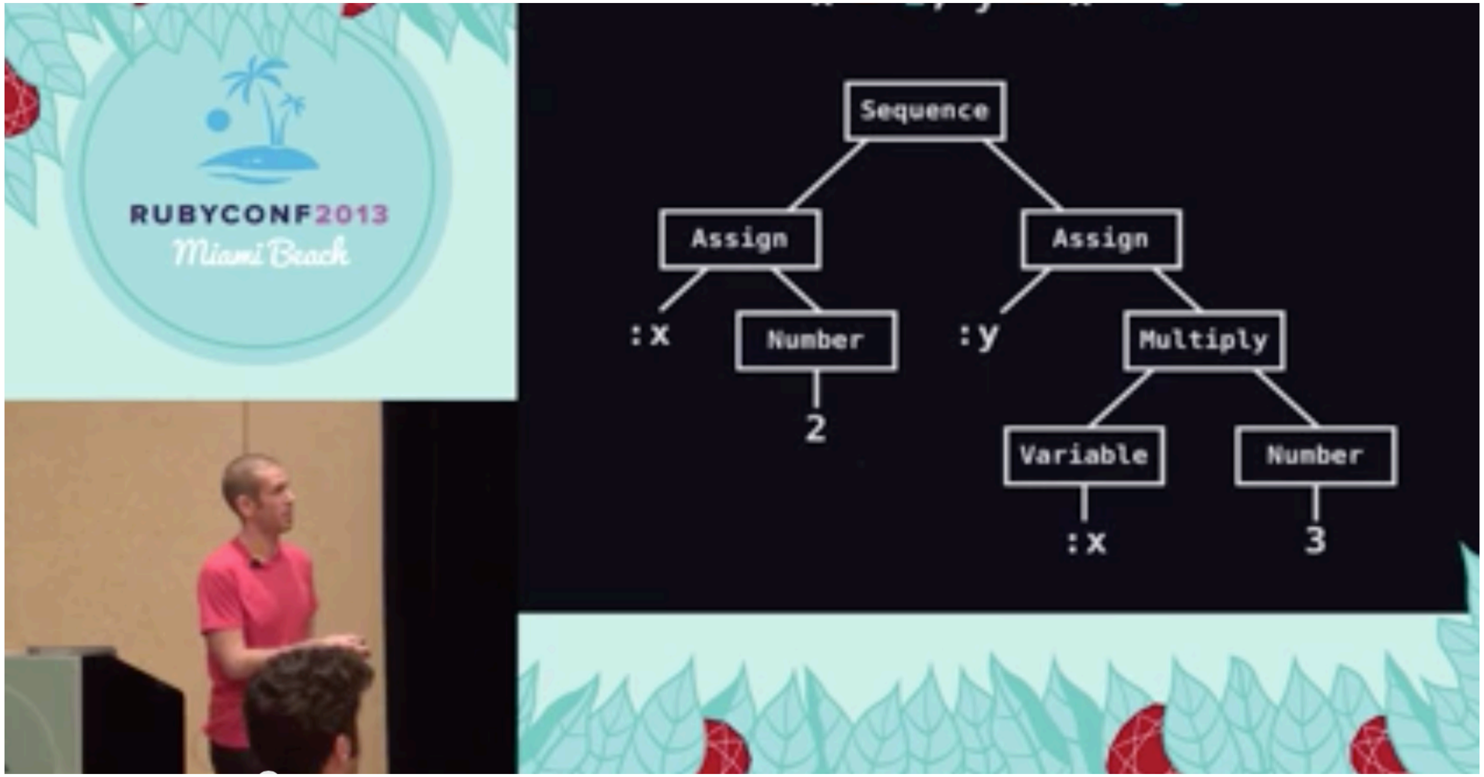
T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In Proceedings of Onward!, 2013.



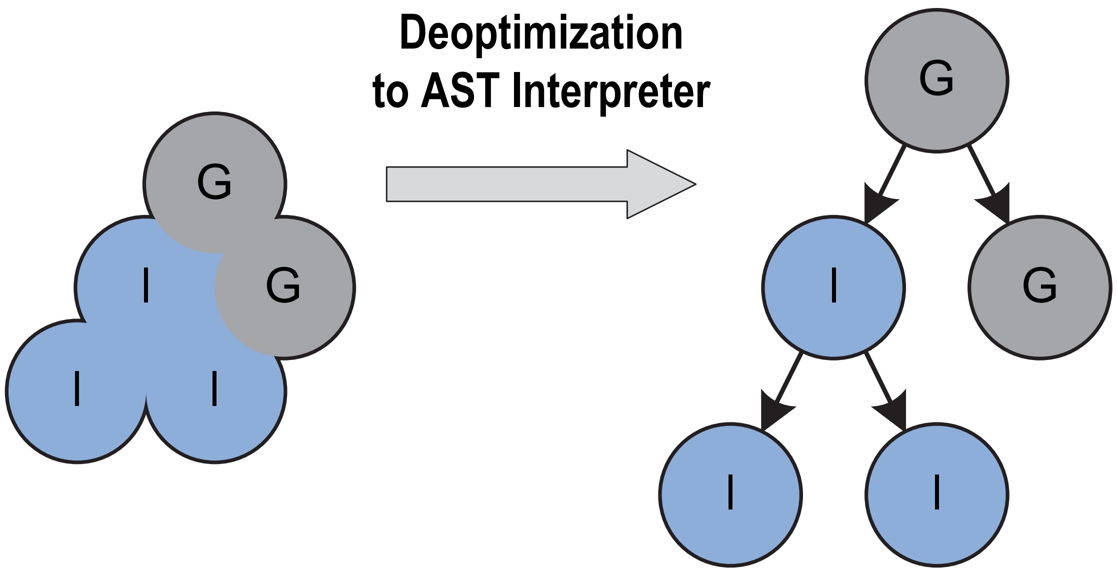
T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In Proceedings of Onward!, 2013.



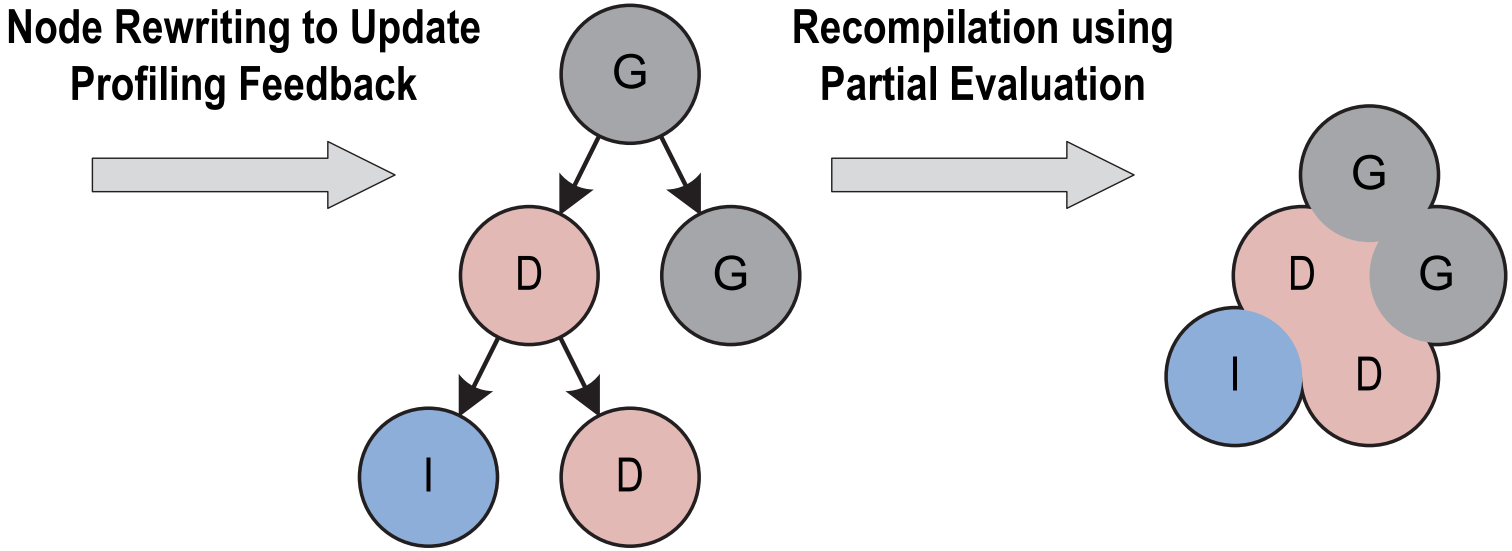
T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In Proceedings of Onward!, 2013.



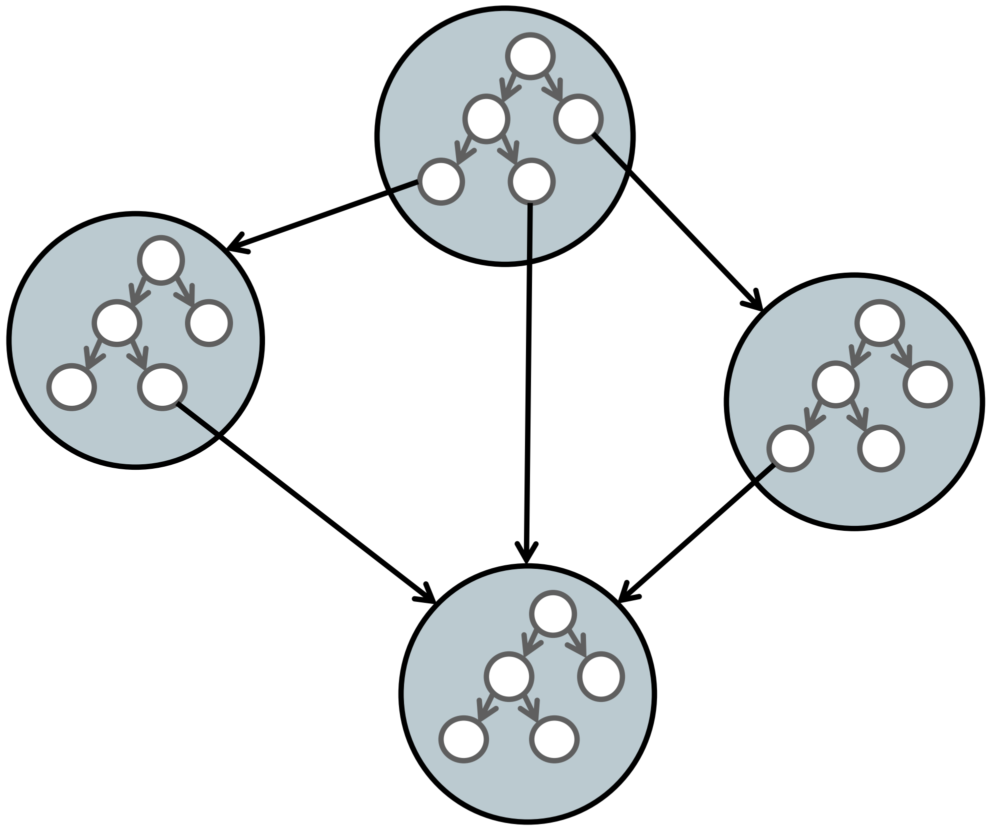
codon.com/compilers-for-free

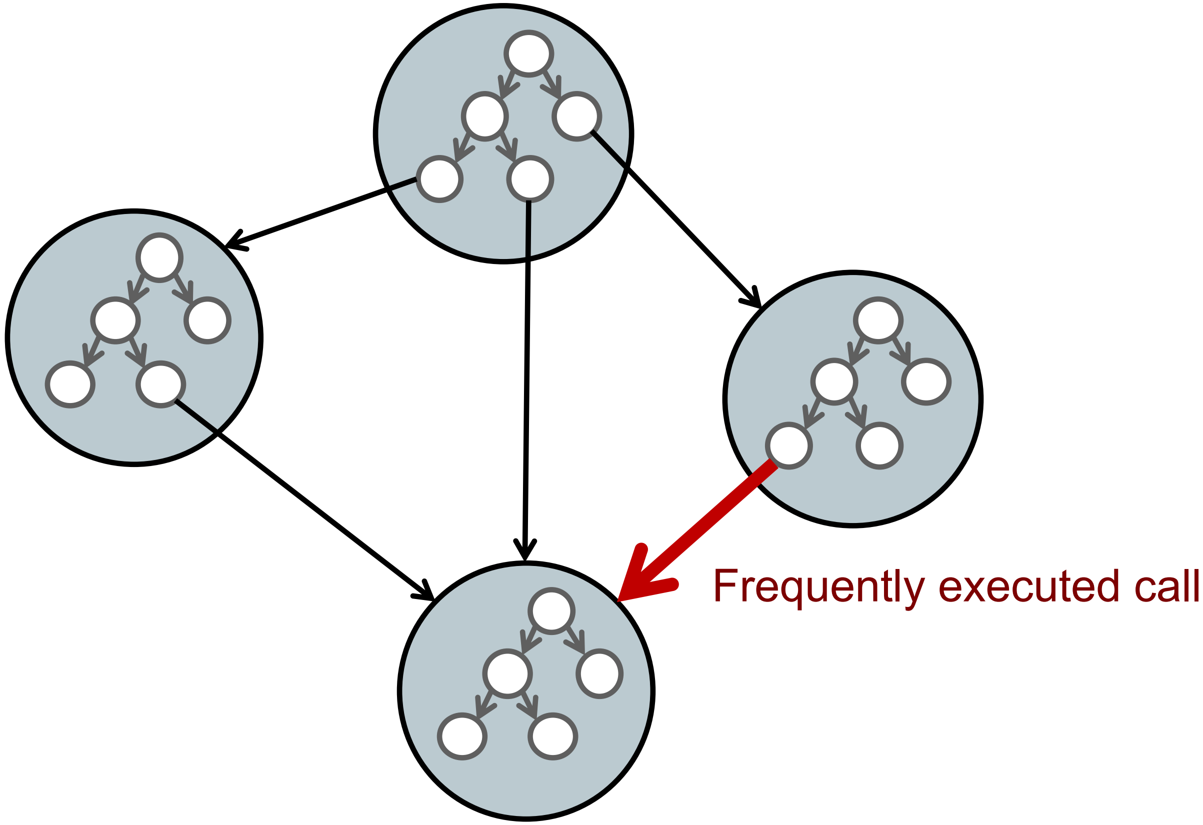


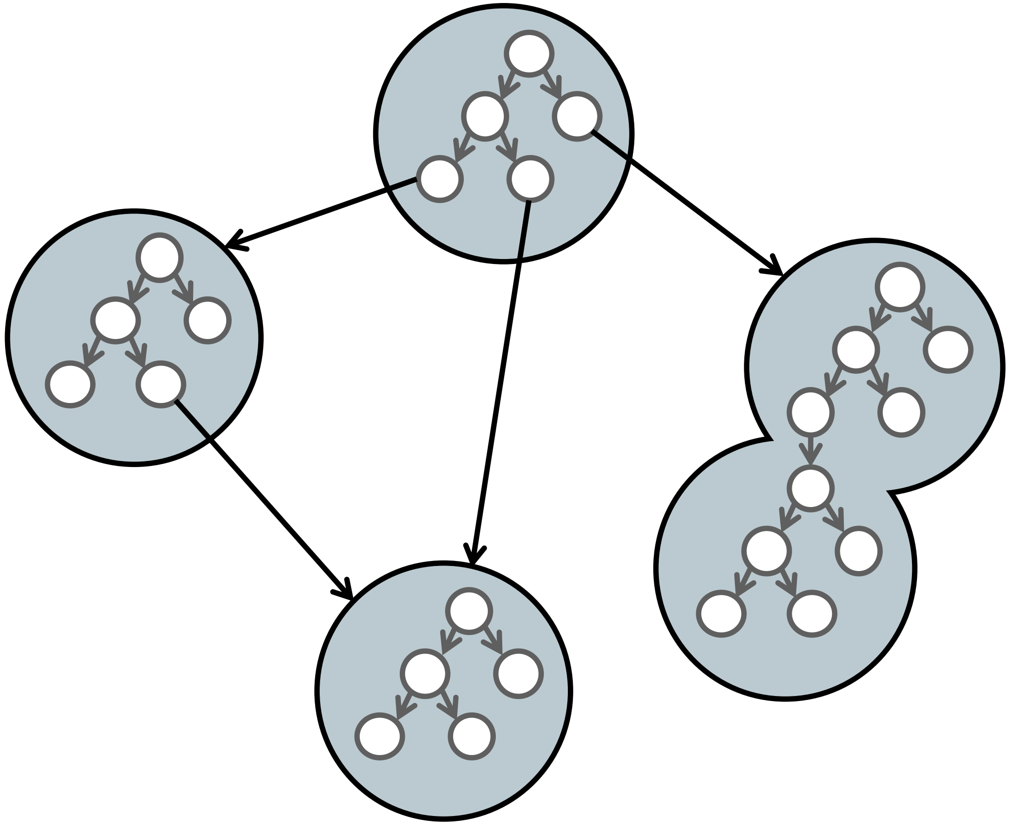
T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In Proceedings of Onward!, 2013.

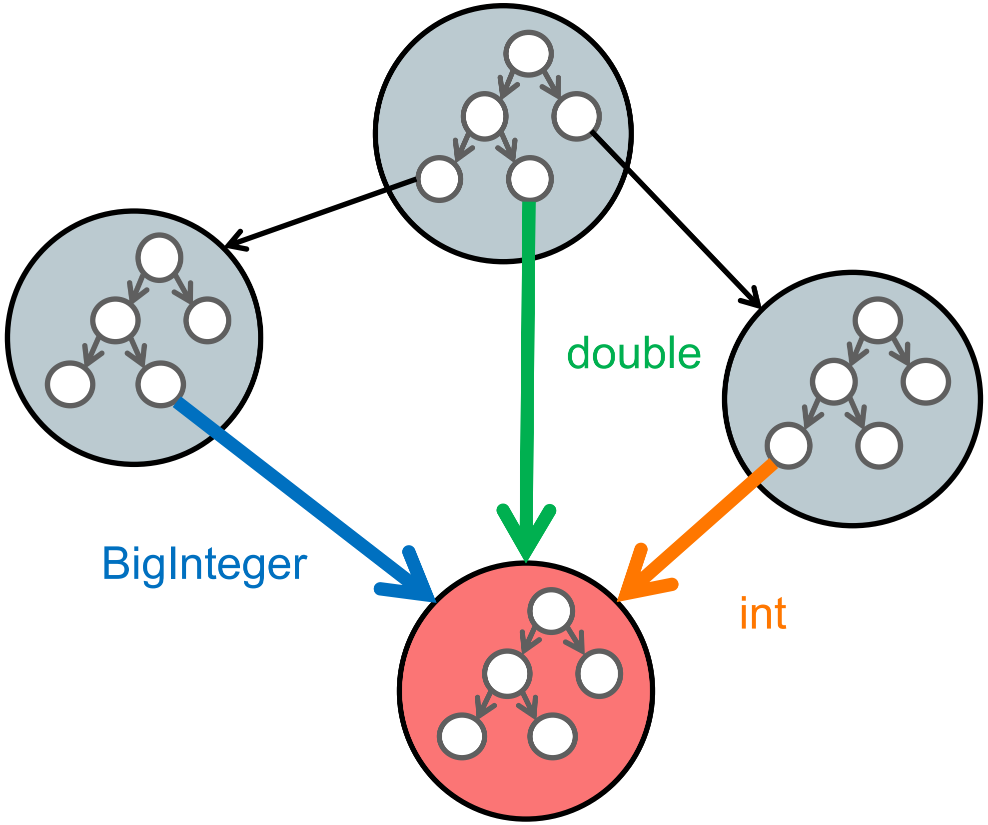


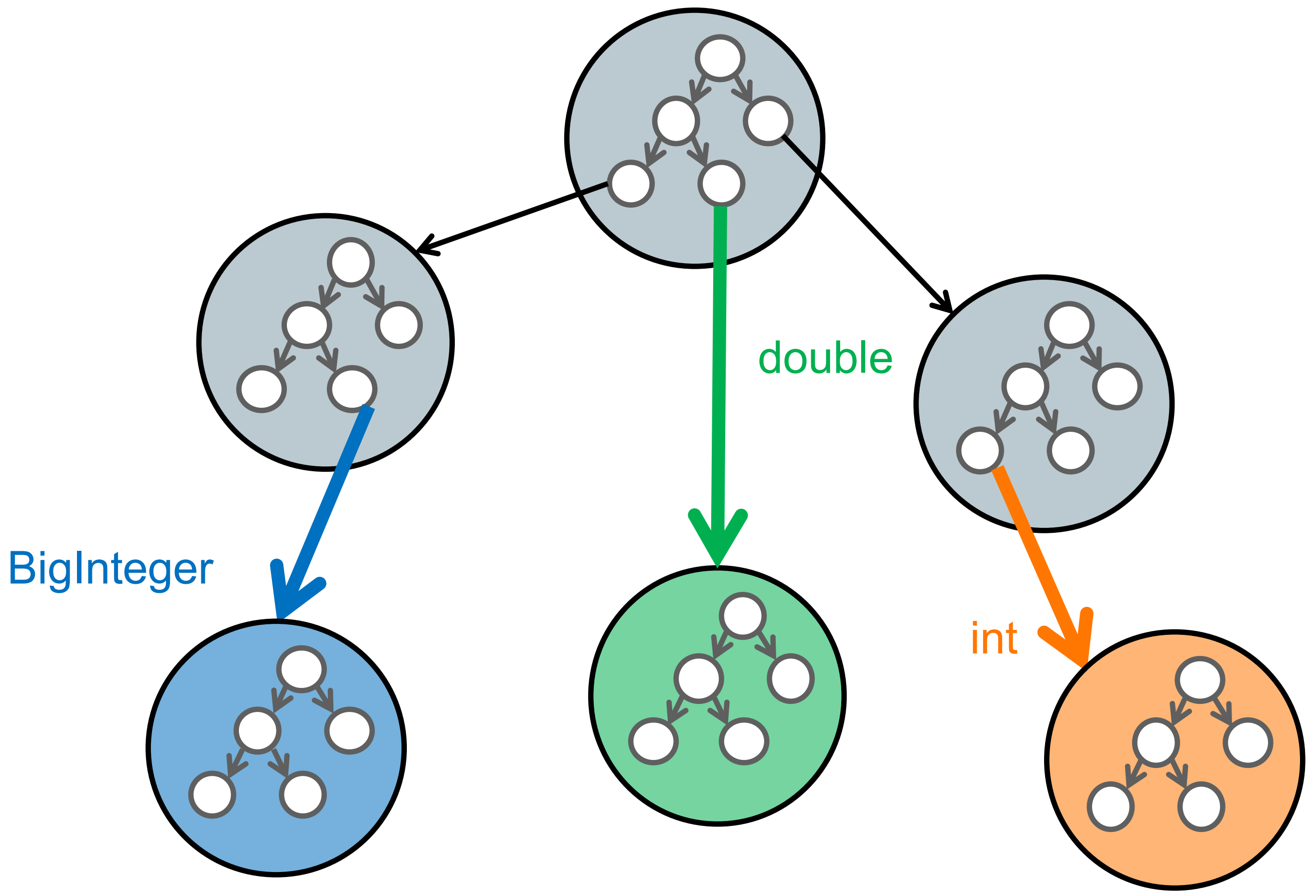
T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In Proceedings of Onward!, 2013.

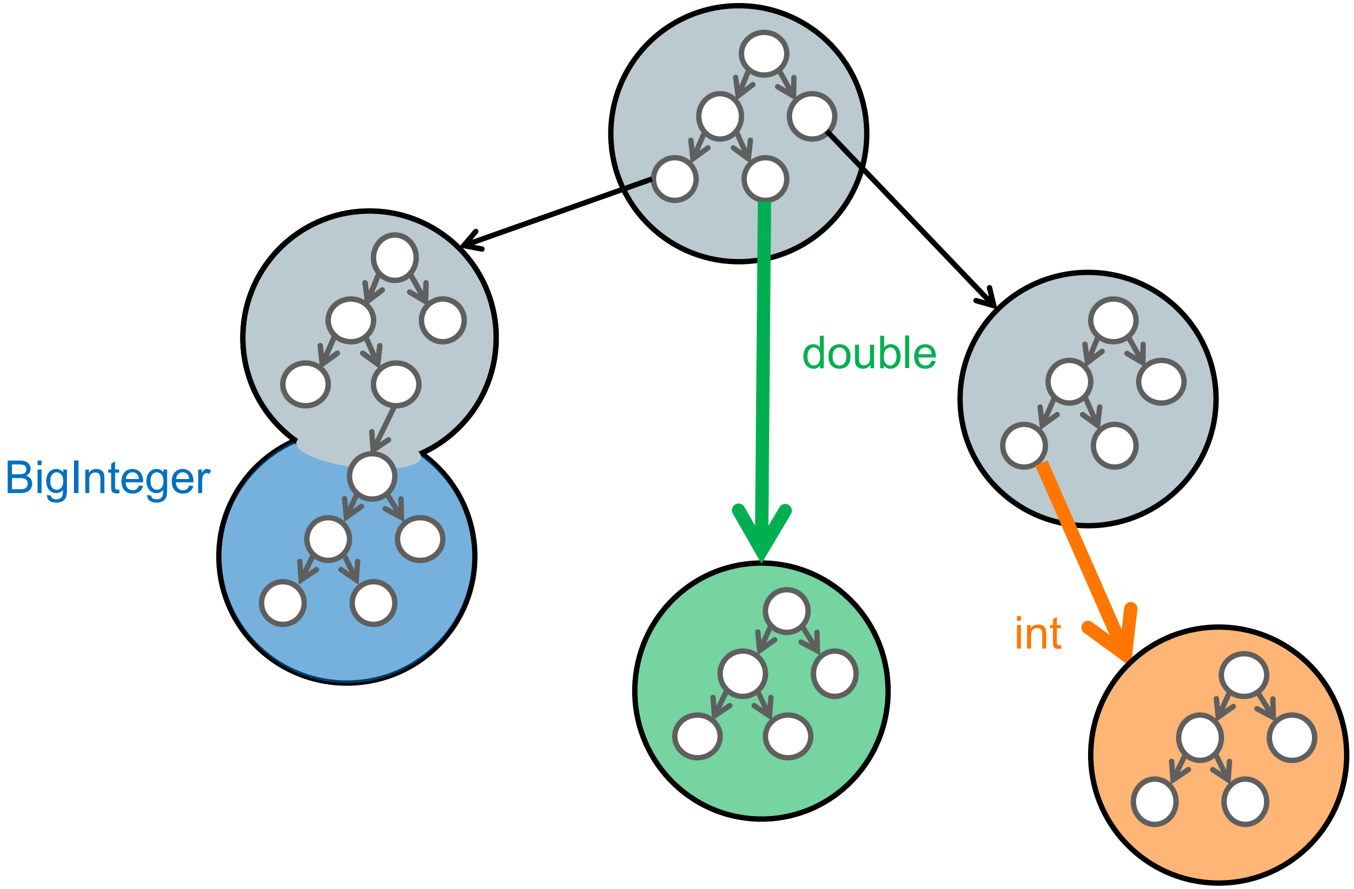



Frequently executed call




double
BigInteger
int


double
BigInteger
int


double
BigInteger
int

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

ORACLE

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

ORACLE

Example: Partial Evaluation and Transfer to Interpreter

```java
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
```

ORACLE

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`

ORACLE

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

- `BranchProfile` to speculate on unlikely branches
  - Benefit: remove code of unlikely code paths

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

- `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

ORACLE

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

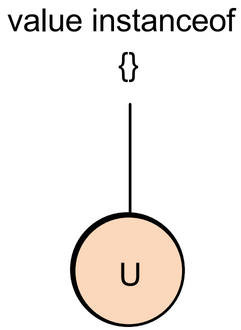
```
class ExampleNode {

  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
    }
  }
}
```

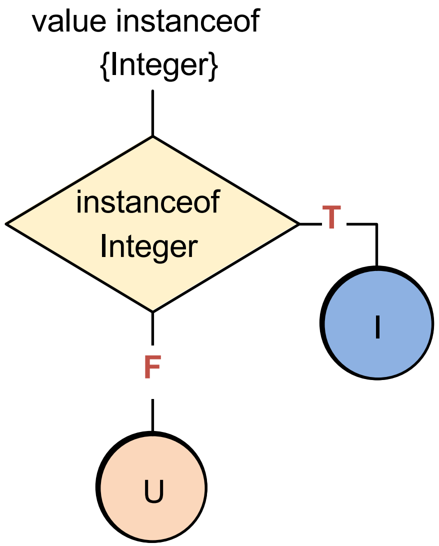
Expected behavior: user does not redefine "+" for integer values

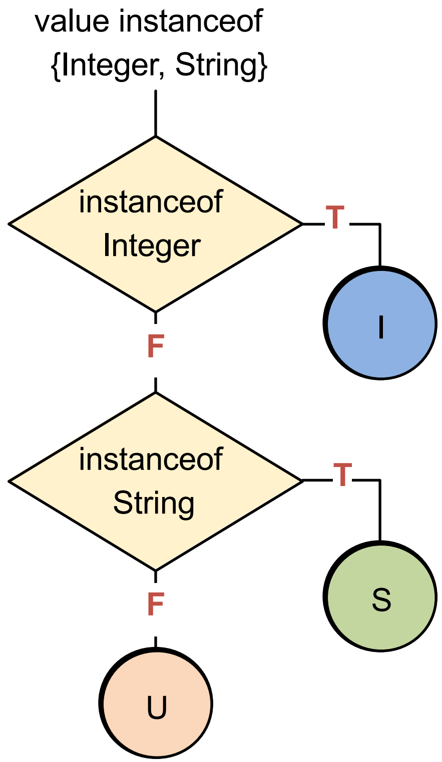
```
void redefineFunction(String name, ...) {
  if (name.equals("+")) {
    addNotRedefined.invalidate()) {
    ...
  }
}
```

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

Specialization







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

ORACLE

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

ORACLE

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`
- 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`
- 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

Version used in this tutorial: tag PLDI_2016

Version used in this tutorial: Graal VM 0.12

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`

ORACLE

If Statement with Profiling

```java
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

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

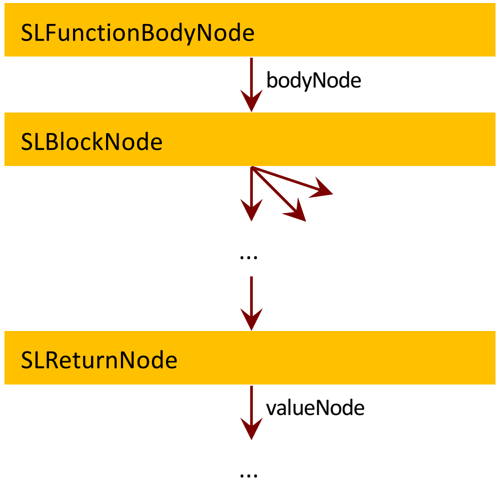
```java
public final class SLFunctionBodyNode extends SLExpressionNode {
  @Child private SLStatementNode bodyNode;
  ...
  public Object executeGeneric(VirtualFrame frame) {
    try {
      bodyNode.executeVoid(frame);
    } catch (SLReturnException ex) {
      return ex.getResult();
    }
    return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

```java
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




```
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

```java
@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

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

ORACLE

`UnexpectedResultException`

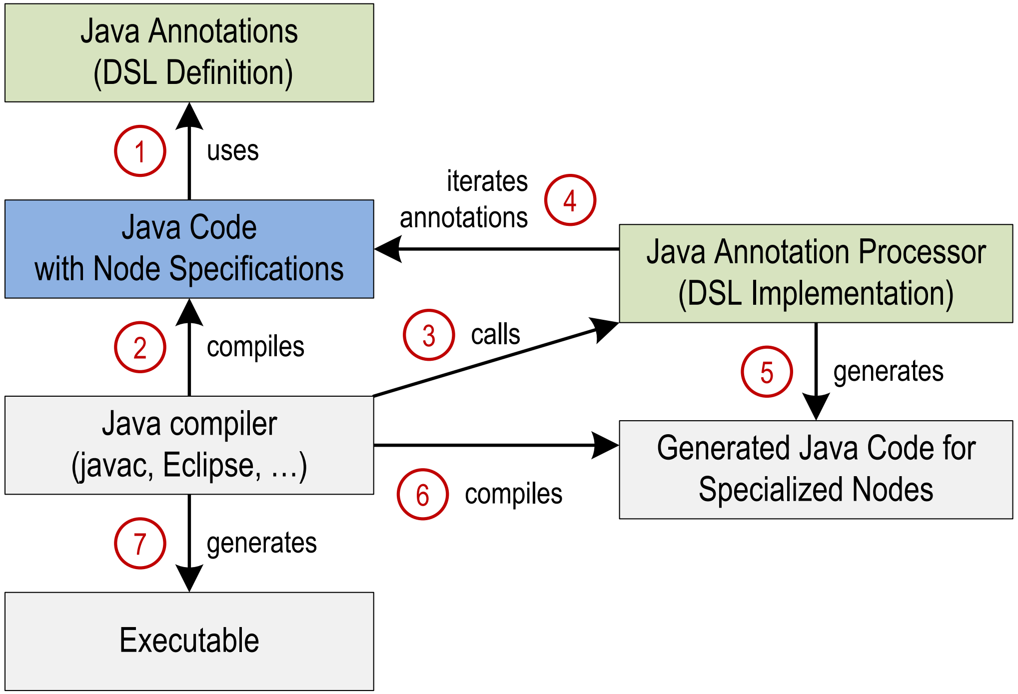
- Type-specialized `execute()` methods have specialized return type
  - Allows primitive return types, to avoid boxing
  - Allows to use the result without type casts
  - Speculation types are stable and the specialization fits

- 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

ORACLE

Truffle DSL Workflow



ORACLE

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

ORACLE

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`

ORACLE

Frame Management

```
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

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

ORACLE

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

ORACLE

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

ORACLE

Slow Path Annotation

```
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

ORACLE

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
```

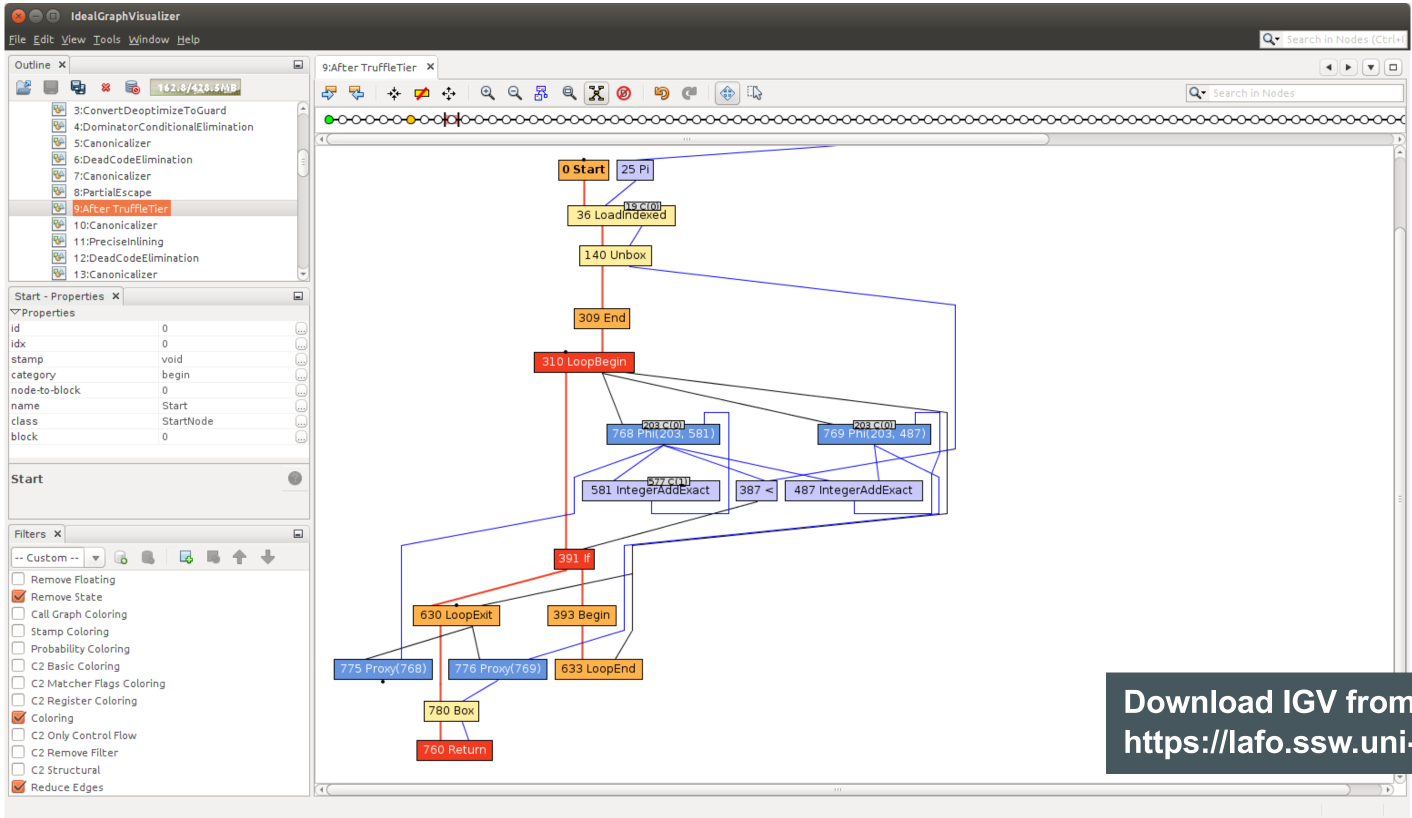
Truffle compilation printing is enabled

Background compilation is disabled

Graph dumping to IGV is enabled

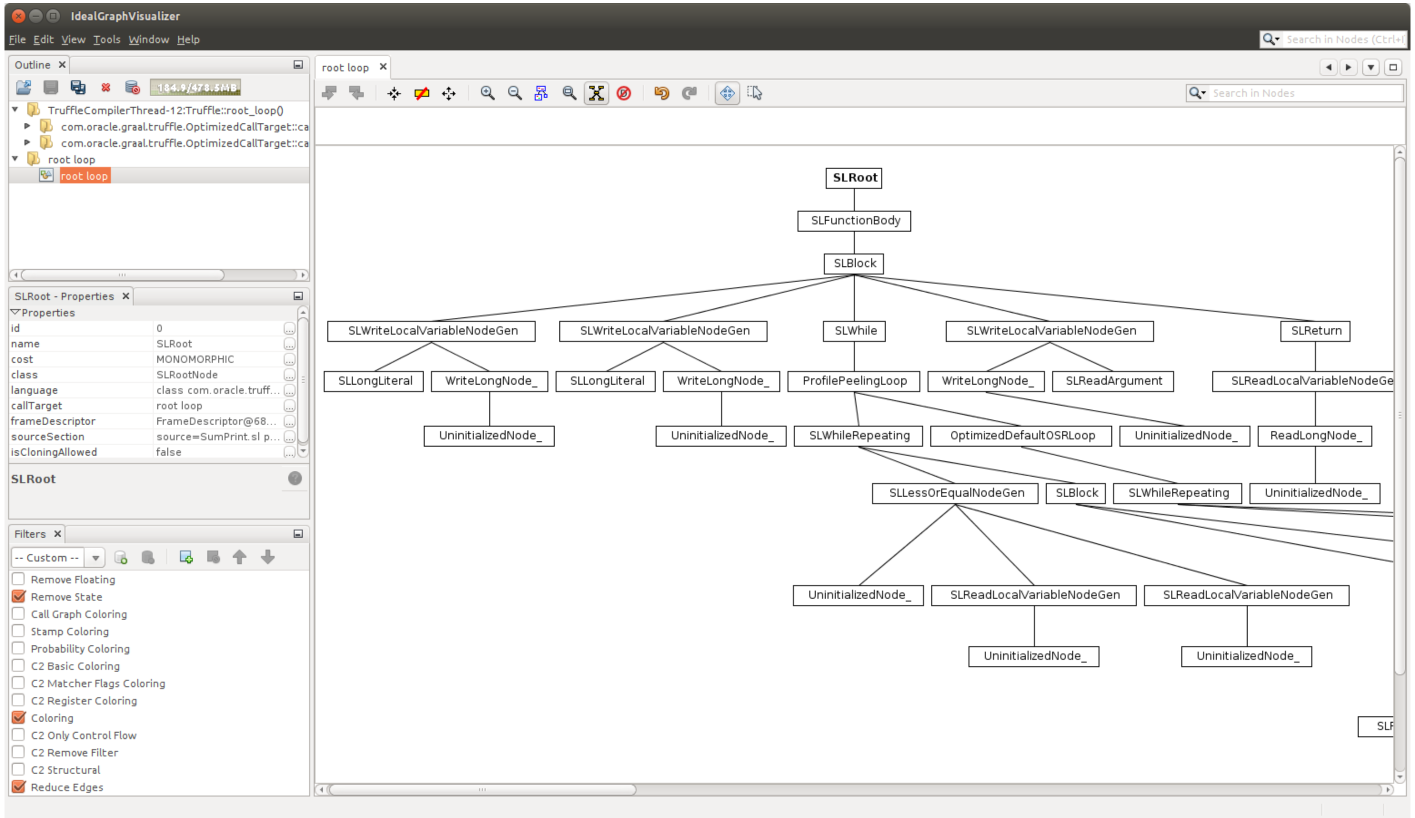
Disassembling is enabled

Visualization Tools: IGV



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

Visualization Tools: IGV



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

ORACLE

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

ORACLE

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

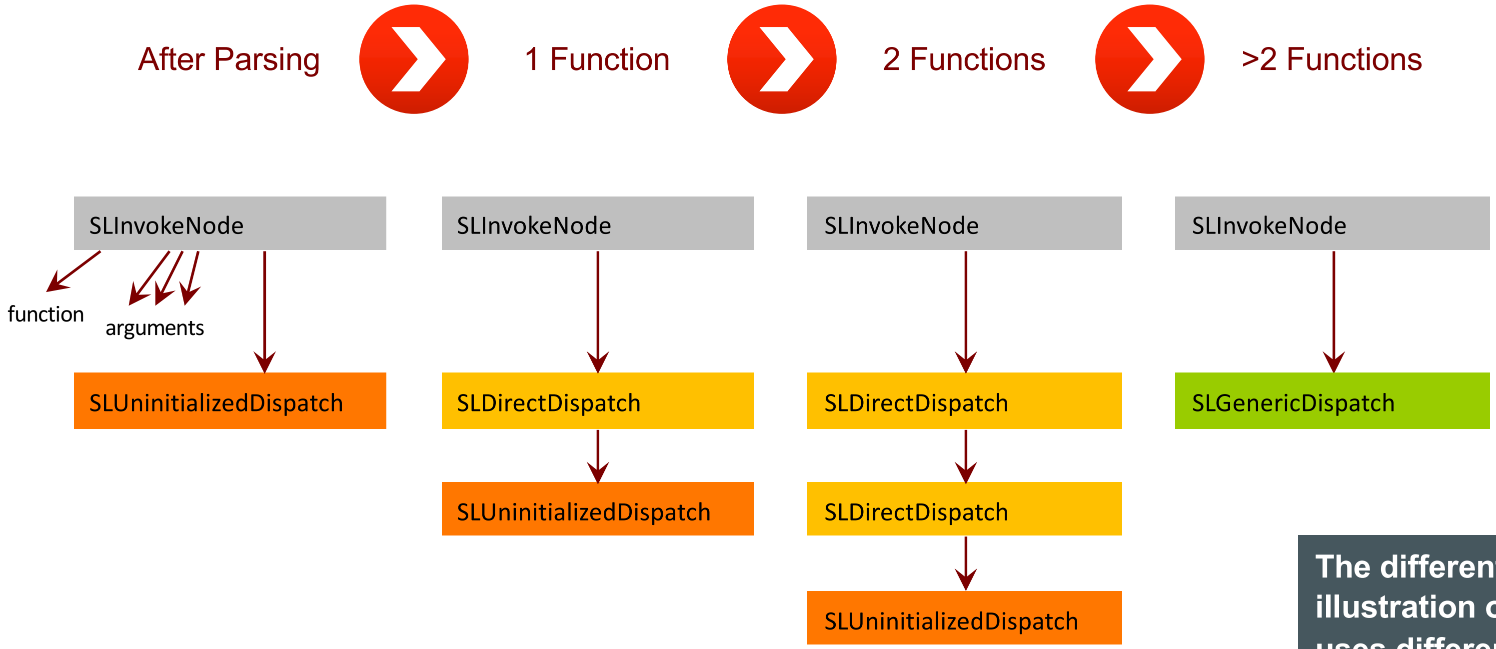
The `operand` is no longer a compile time constant

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



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

```
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

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);
```

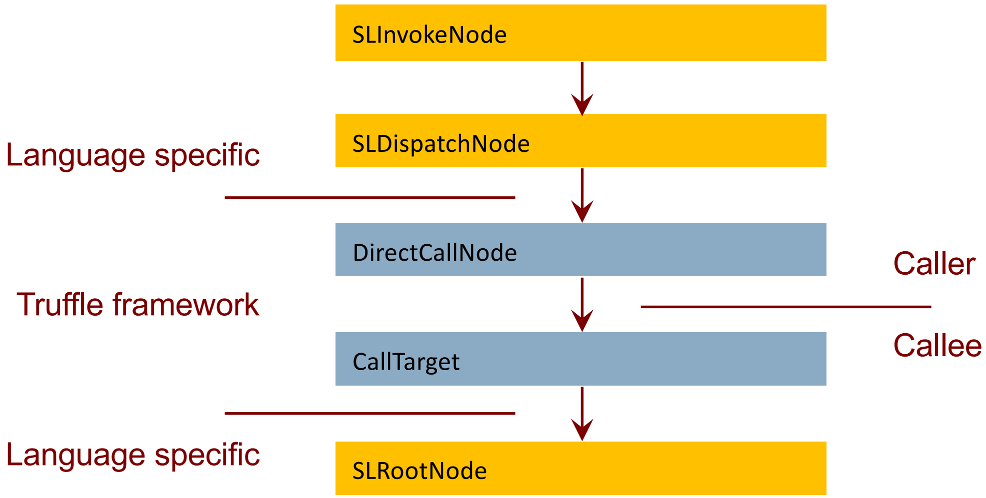
Code that is compiled to a no-op is marked ~~strikethrough~~

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`

ORACLE

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)

```
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")
```

ORACLE

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!

ORACLE

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`

ORACLE

Object Allocation

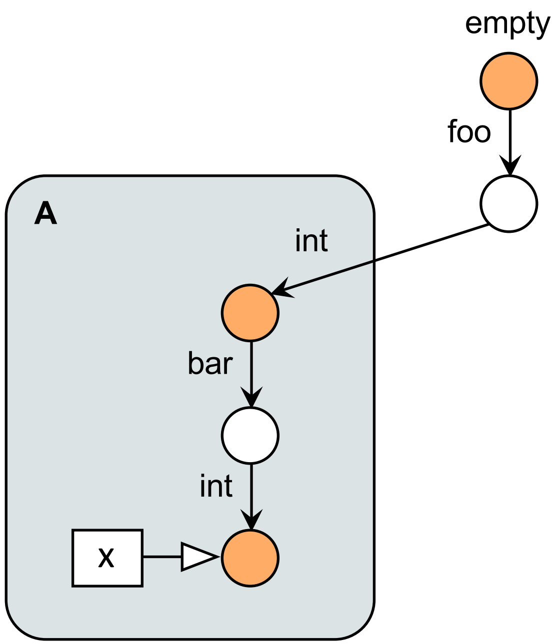
```
public final class SLContext extends ExecutionContext {
  private static final Layout LAYOUT = Layout.createLayout();

  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();
}
```

ORACLE

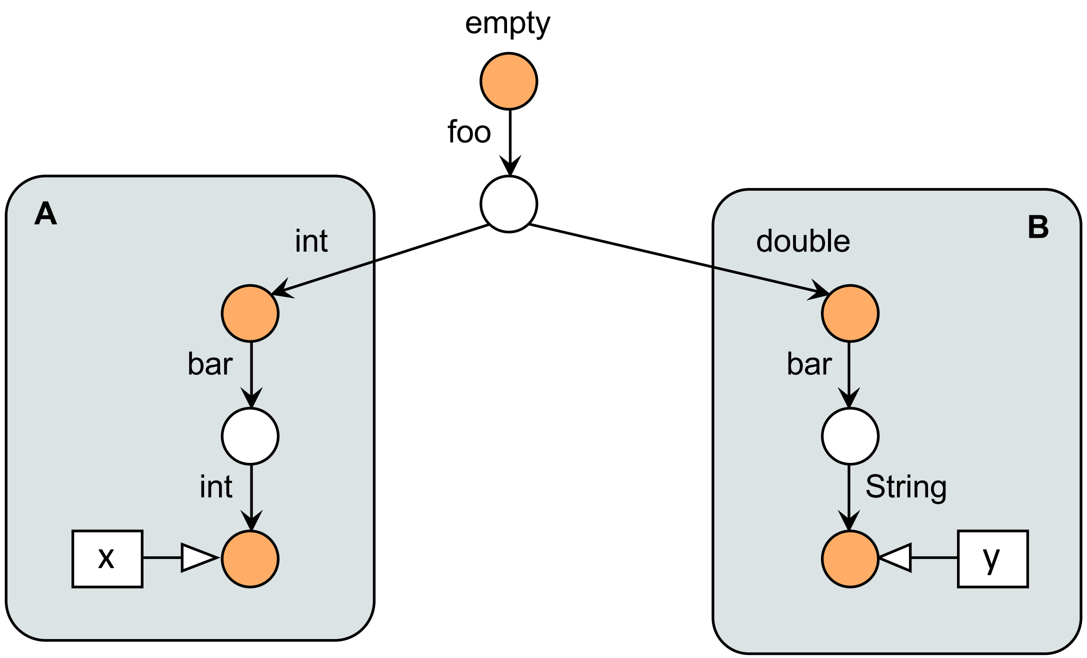
Object Layout Transitions (1)

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



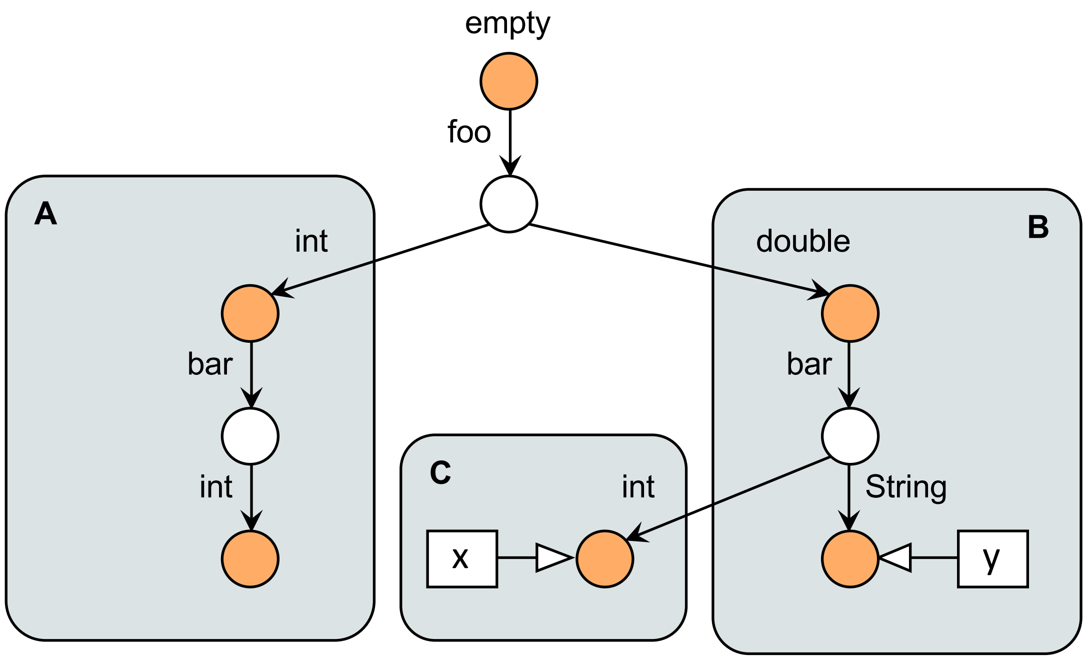
Object Layout Transitions (2)

```
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 †Oracle Labs
{woess, christian.humer, moessenboeck}@ssw.jku.at {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

```
@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;
}
```

```
@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);
}
```

ORACLE

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!

ORACLE

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

ORACLE

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

```
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

Polyglot

Language Registration

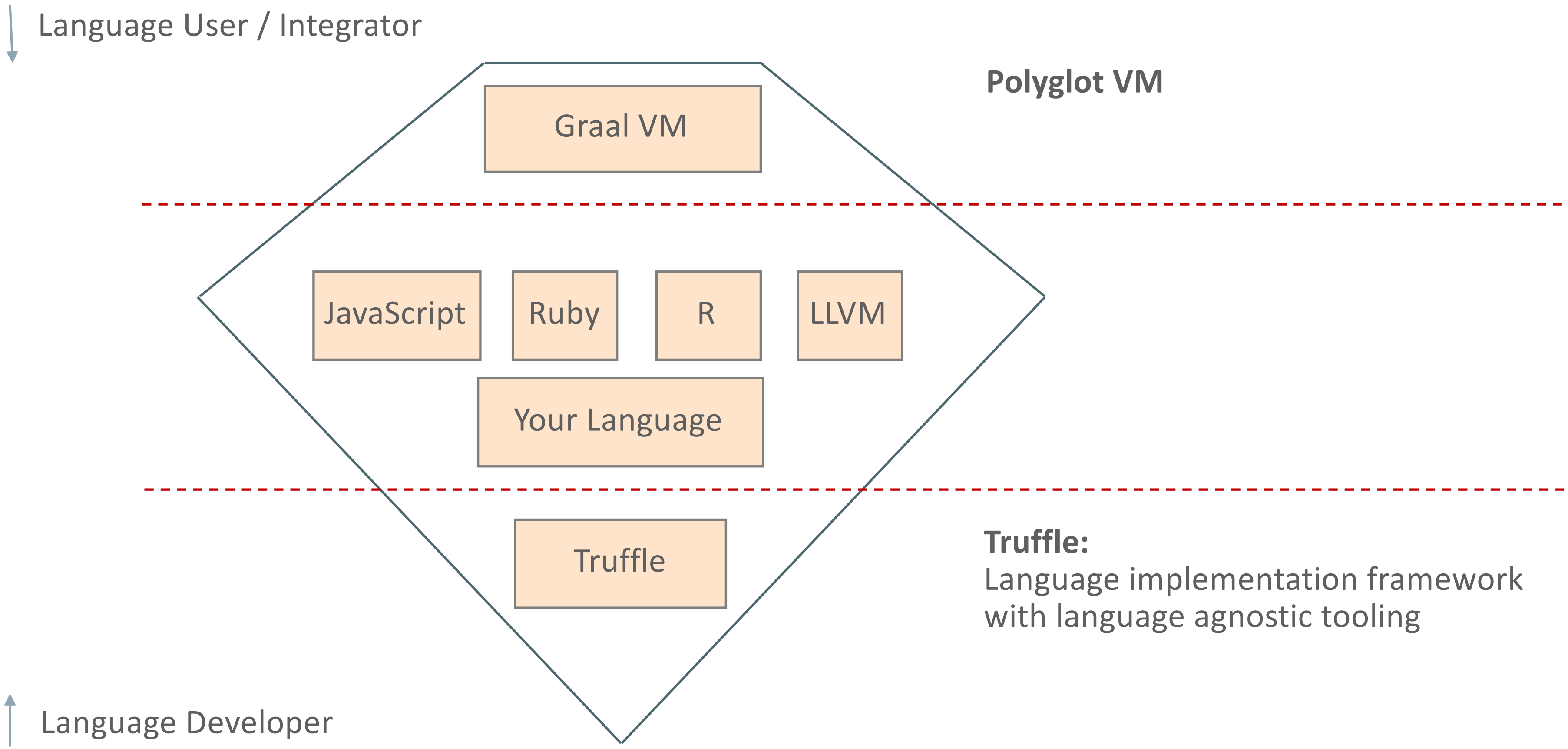
```
public final class SLMain {

  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 { ... }
```

The Polyglot Diamond



Graal VM Multi-Language Shell

Add a vector of numbers using three languages:

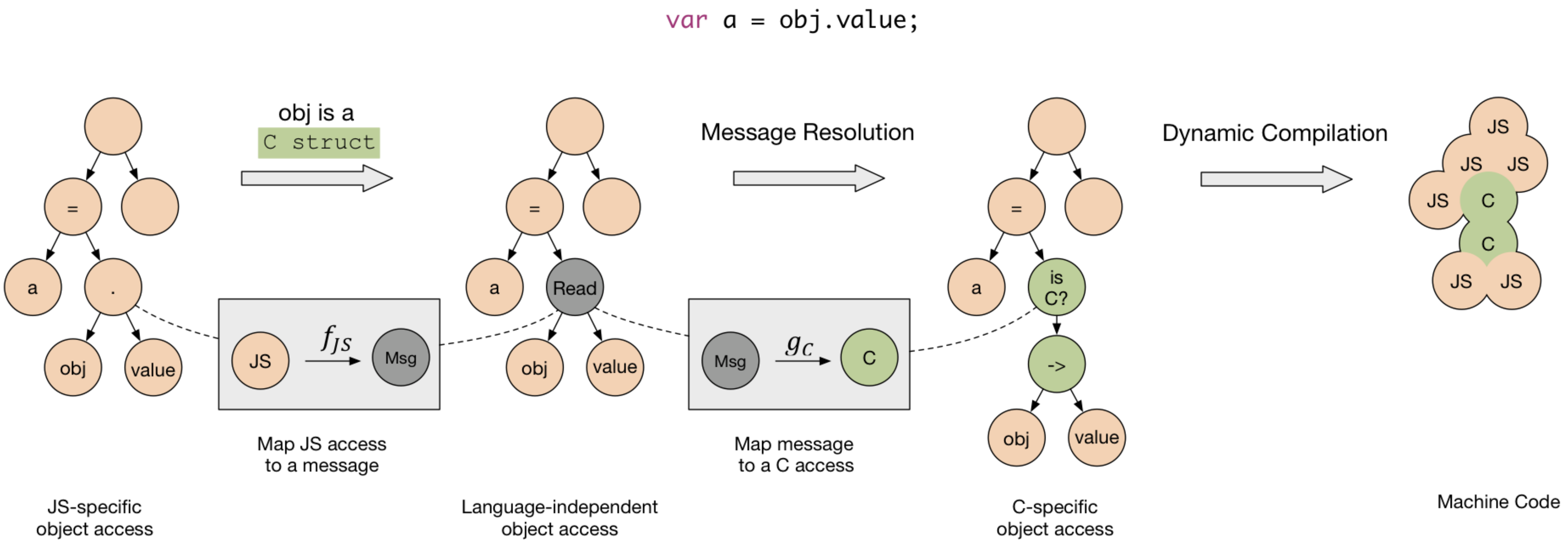
```
Ruby>
def rubyadd(a, b)
  a + b;
end
Truffle::Interop.export_method(:rubyadd);

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

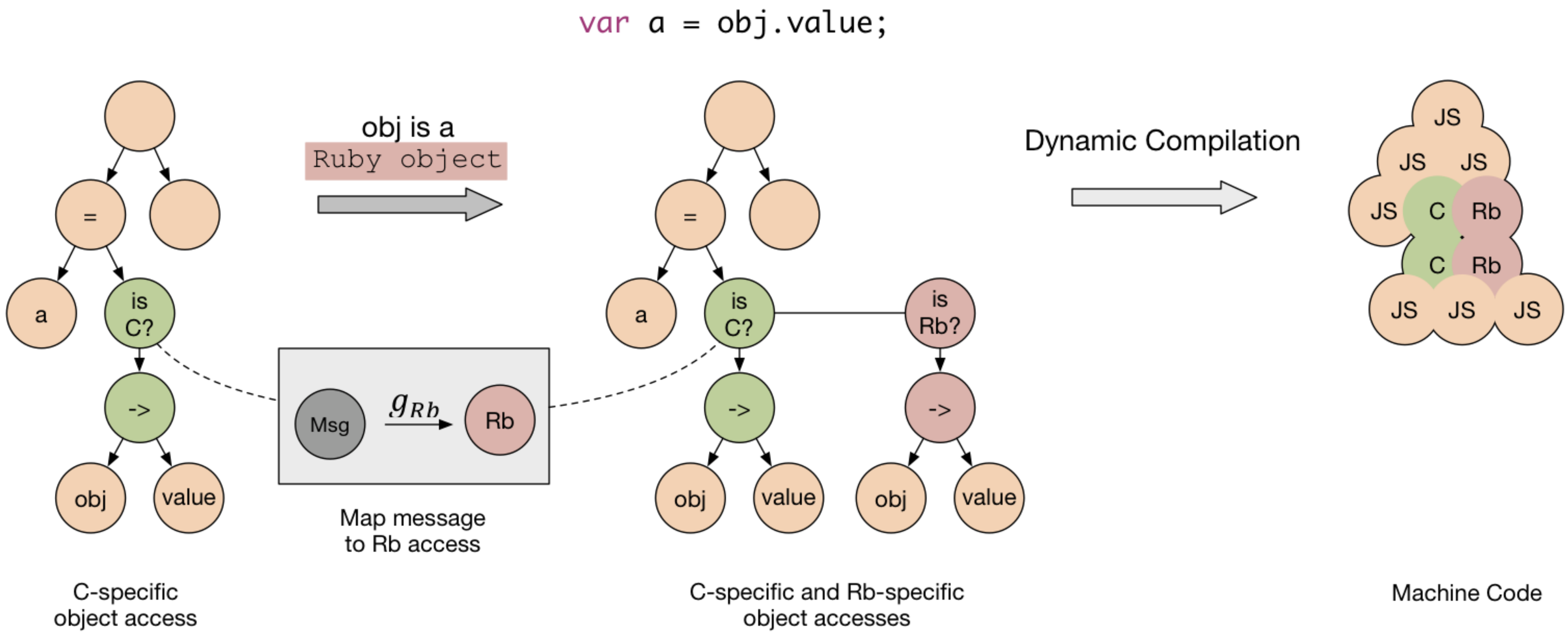
Start `bin/graalvm`

Explicit export and import of symbols (methods)

High-Performance Language Interoperability (1)



High-Performance Language Interoperability (2)



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();
  }
}
```

ORACLE

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!

ORACLE

Polyglot Example: Mixing Ruby and JavaScript

```
14 + 2
```

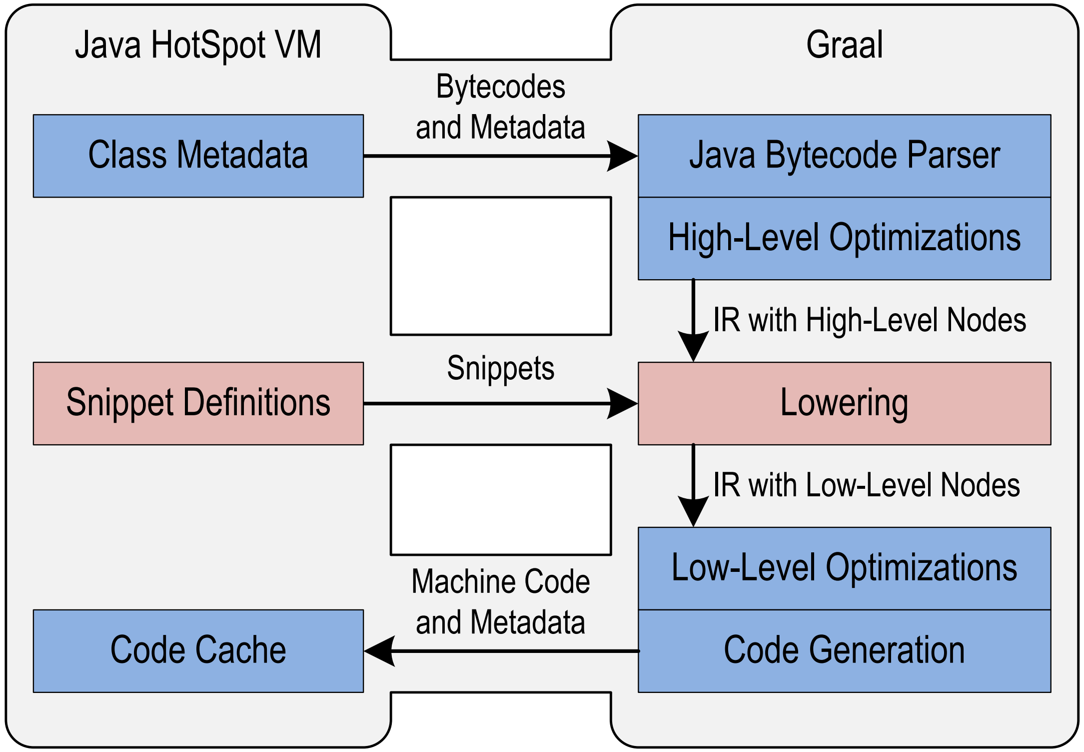
```
ExecJS.eval('14 + 2')
```

```
$ 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
```

Graal

Compiler-VM Separation



Basic Properties

- Two interposed directed graphs
  - Control flow graph: Control flow edges point “downwards” in graph
  - Data flow graph: Data flow edges point “upwards” in graph
- Floating nodes
  - Nodes that can be scheduled freely are not part of the control flow graph
  - Avoids unnecessary restrictions of compiler optimizations
- Graph edges specified as annotated Java fields in node classes
  - Control flow edges: @Successor fields
  - Data flow edges: @Input fields
  - Reverse edges (i.e., predecessors, usages) automatically maintained by Graal
- Always in Static Single Assignment (SSA) form
- Only explicit and structured loops
  - Loop begin, end, and exit nodes
- Graph visualization tool: “Ideal Graph Visualizer”, start using “./mx.sh igv”

ORACLE

IR Example: Defining Nodes

```
public abstract class BinaryNode ... {
  @Input protected ValueNode x;
  @Input protected ValueNode y;
}
```

@Input fields: data flow

```
public class IfNode ... {
  @Successor BeginNode trueSuccessor;
  @Successor BeginNode falseSuccessor;
  @Input(InputType.Condition) LogicNode condition;
  protected double trueSuccessorProbability;
}
```

@Successor fields: control flow

Fields without annotation: normal data properties

```
public abstract class Node ... {
  public NodeClassIterable inputs() { ... }
  public NodeClassIterable successors() { ... }

  public NodeIterable<Node> usages() { ... }
  public Node predecessor() { ... }
}
```

Base class allows iteration of all inputs / successors

Base class maintains reverse edges: usages / predecessor

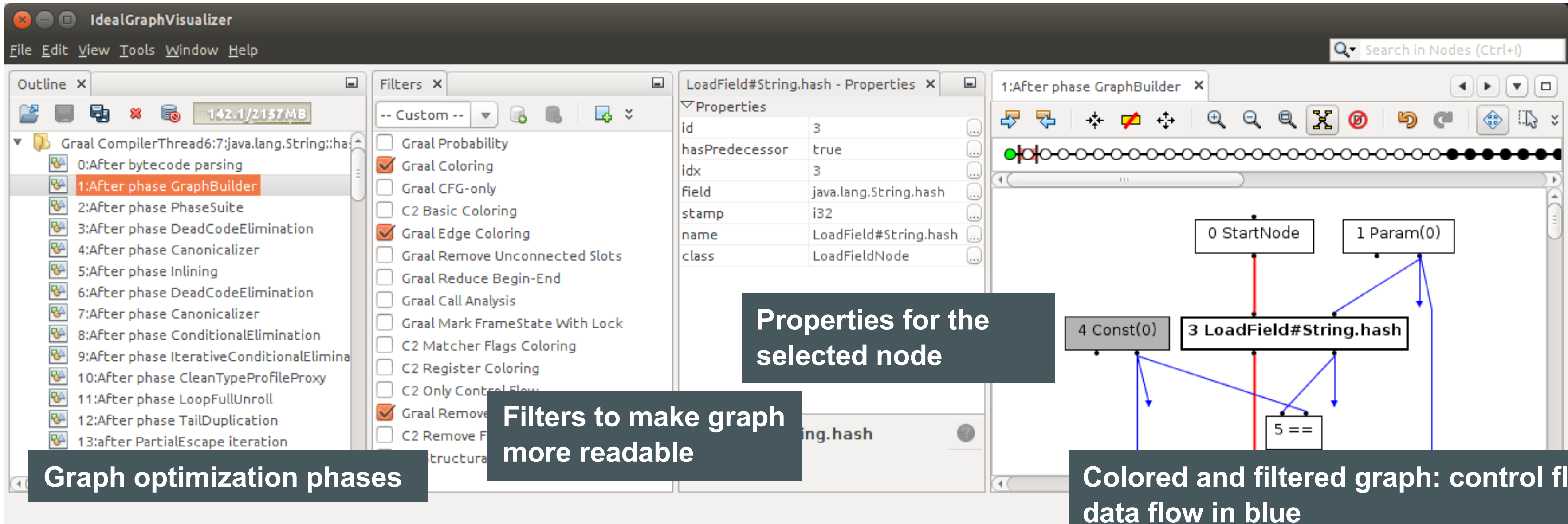
Design invariant: a node has at most one predecessor

IR Example: Ideal Graph Visualizer

Start the Graal VM with graph dumping enabled

```
$ ./mx.sh igv &
$ ./mx.sh unittest -G:Dump= -G:MethodFilter=String.hashCode GraalTutorial#testStringHashCode
```

Test that just compiles `String.hashCode()`



Properties for the selected node

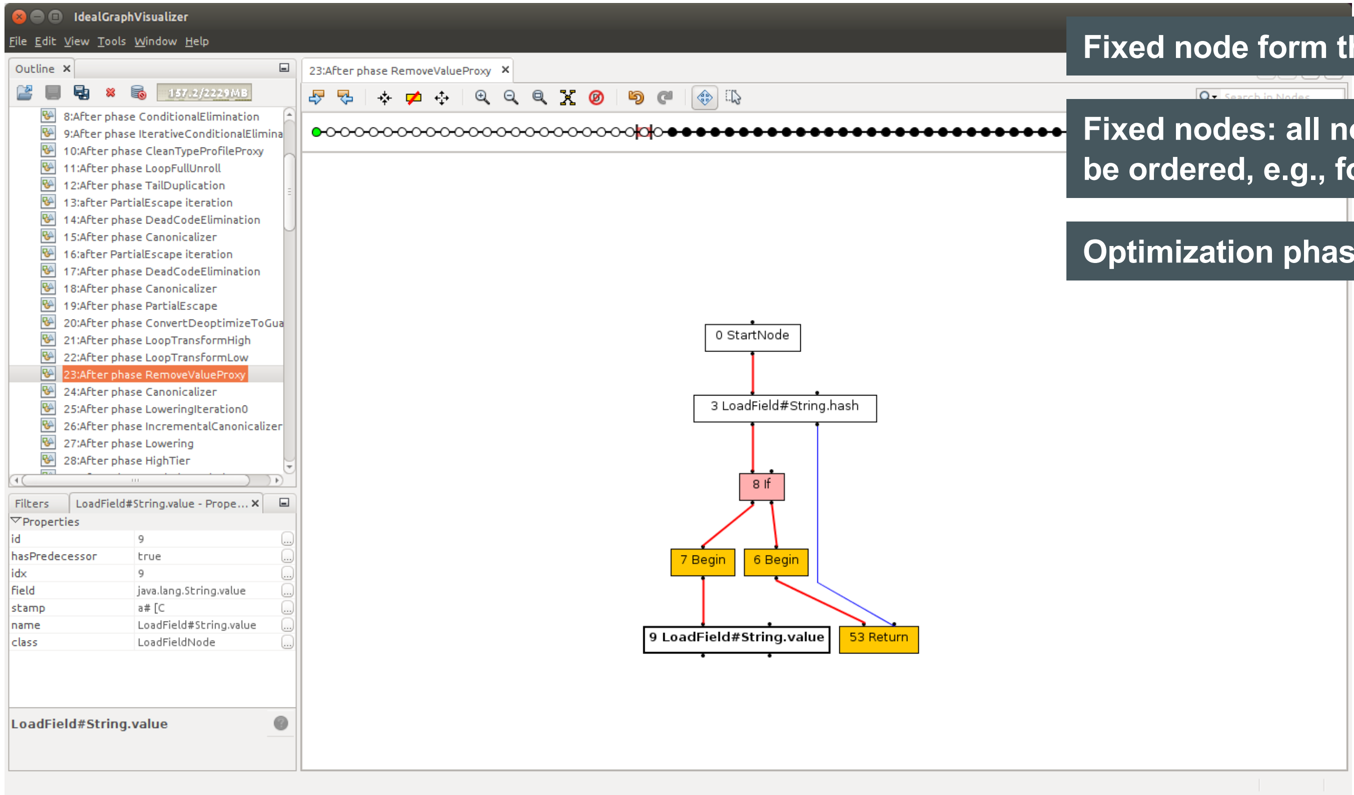
Filters to make graph more readable

Graph optimization phases

Colored and filtered graph: control flow in red, data flow in blue

ORACLE

IR Example: Control Flow

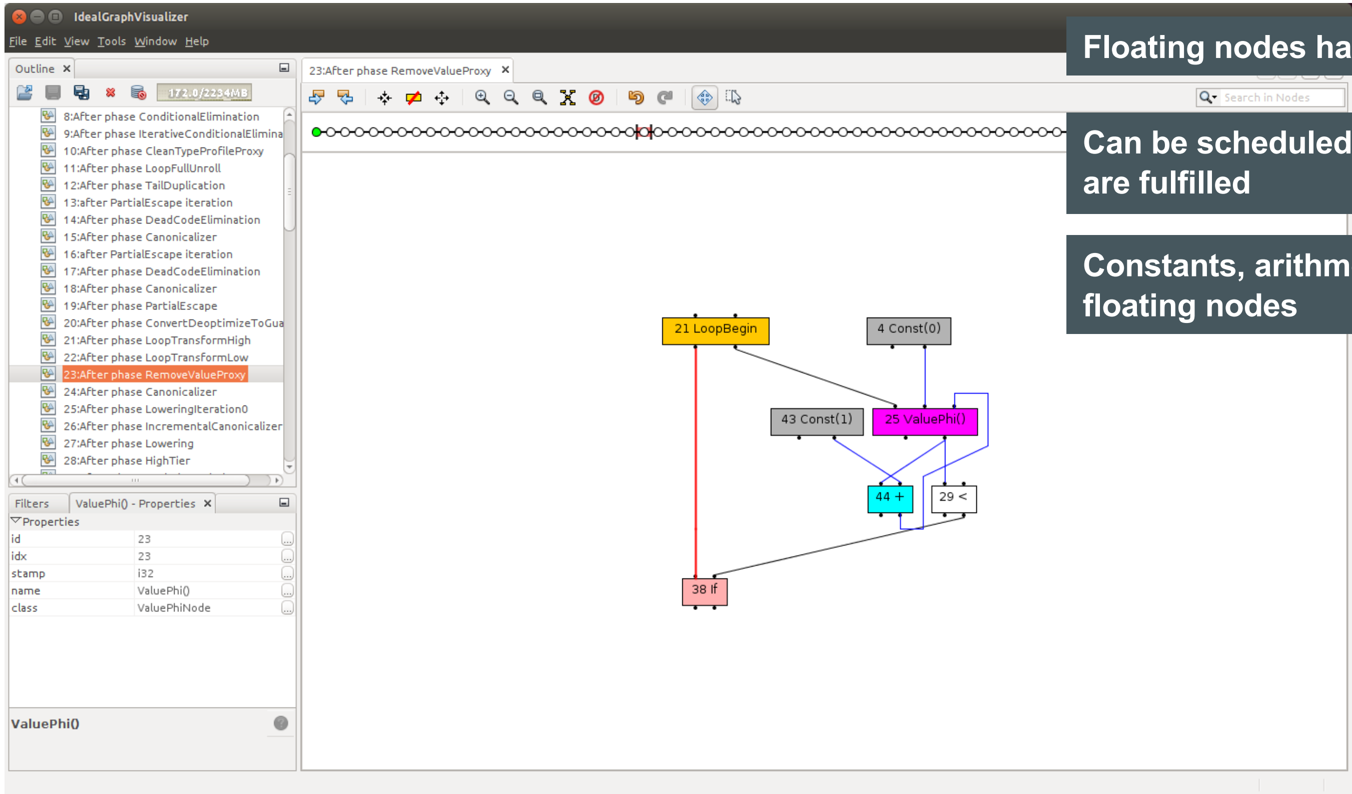


Fixed node form the control flow graph

Fixed nodes: all nodes that have side effects and need to be ordered, e.g., for Java exception semantics

Optimization phases can convert fixed to floating nodes

IR Example: Floating Nodes

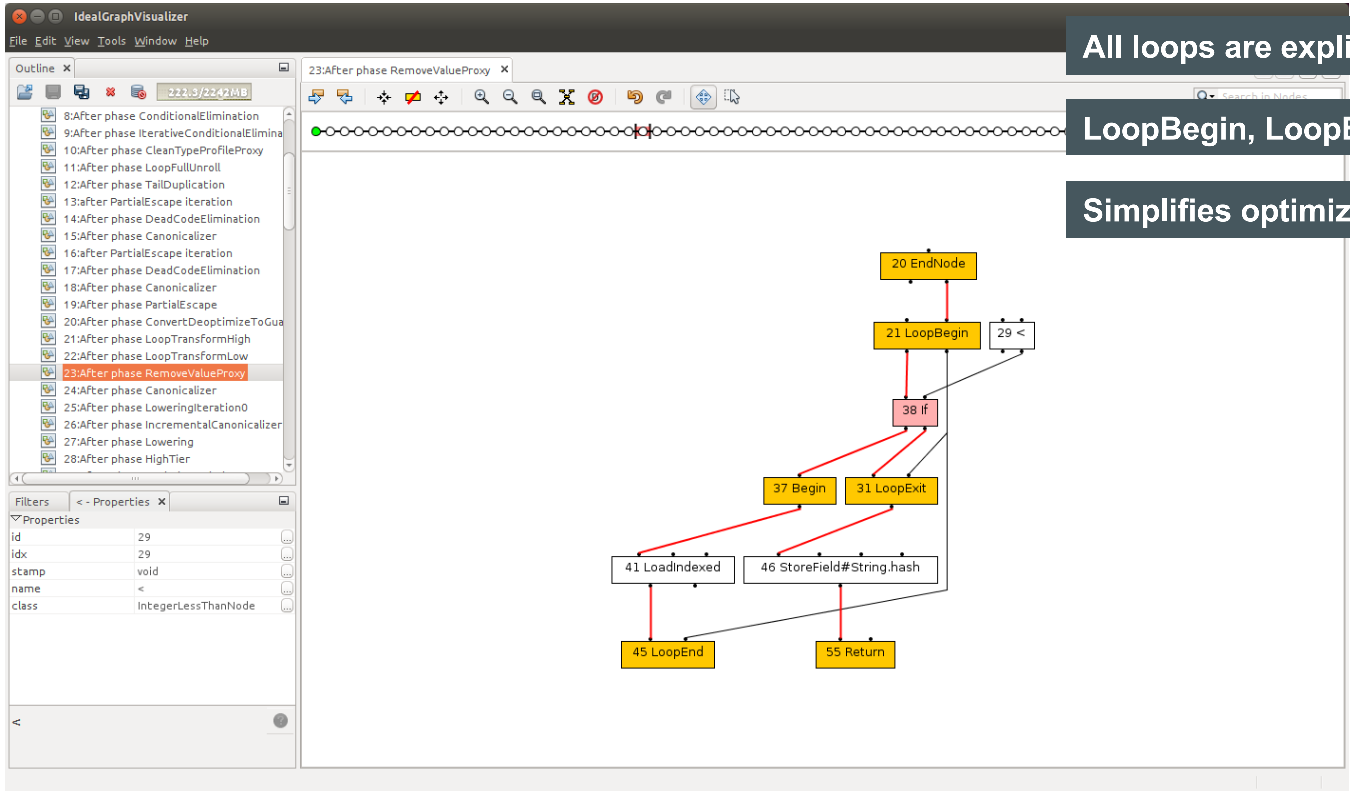


Floating nodes have no control flow dependency

Can be scheduled anywhere as long as data dependencies are fulfilled

Constants, arithmetic functions, phi functions, … are floating nodes

IR Example: Loops



All loops are explicit and structured

LoopBegin, LoopEnd, LoopExit nodes

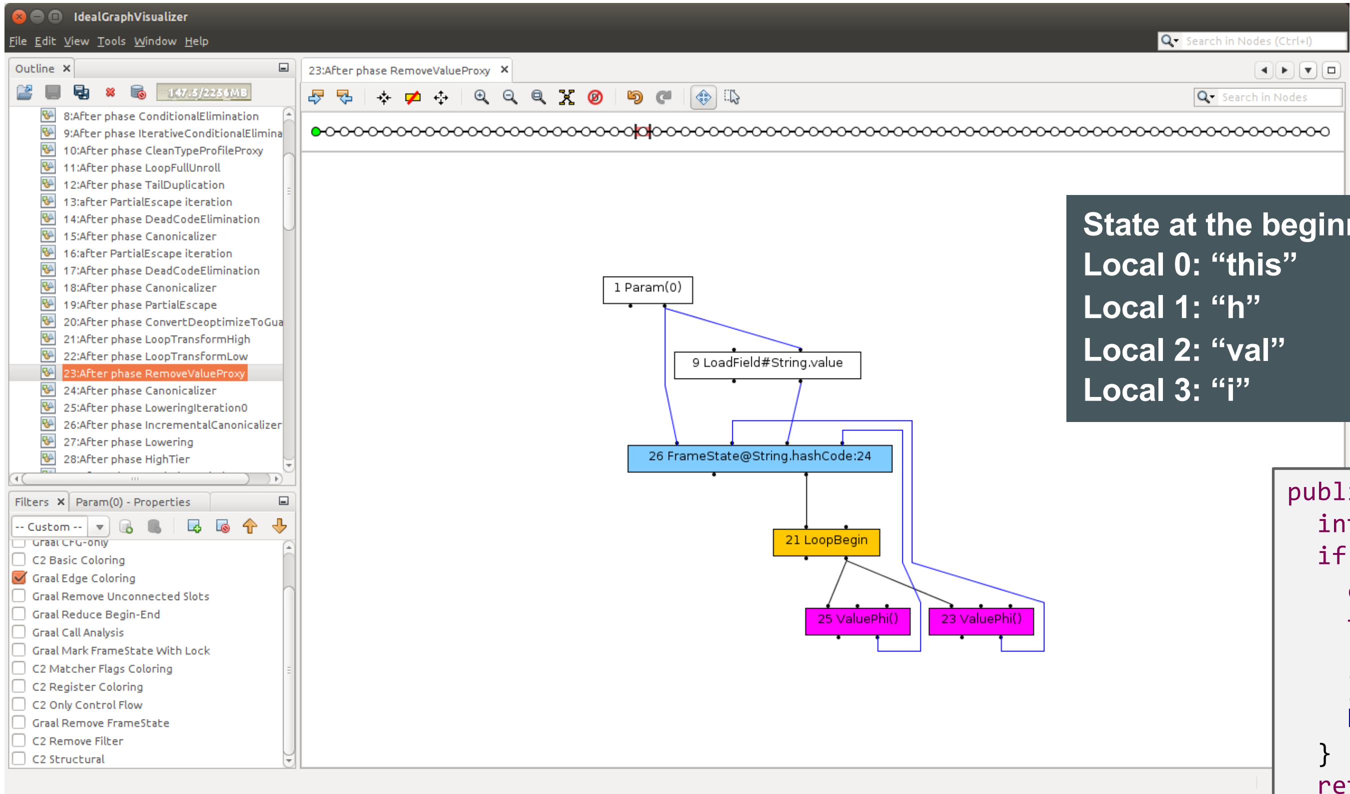
Simplifies optimization phases

FrameState

- Speculative optimizations require deoptimization
  - Restore Java interpreter state at safepoints
  - Graal tracks the interpreter state throughout the whole compilation
    - FrameState nodes capture the state of Java local variables and Java expression stack
    - And: method + bytecode index

- Method inlining produces nested frame states
  - `FrameState` of callee has `@Input outerFrameState`
  - Points to `FrameState` of caller

IR Example: Frame States



State at the beginning of the loop:
Local 0: "this"
Local 1: "h"
Local 2: "val"
Local 3: "i"

```
public int hashCode() {
  int h = hash;
  if (h == 0 && value.length > 0) {
    char val[] = value;
    for (int i = 0; i < value.length; i++) {
      h = 31 * h + val[i];
    }
    hash = h;
  }
  return h;
}
```

Important Optimizations

- Constant folding, arithmetic optimizations, strength reduction, ...
  - `CanonicalizerPhase`
  - Nodes implement the interface `Canonicalizeable`
  - Executed often in the compilation pipeline
  - Incremental canonicalizer only looks at new / changed nodes to save time

- Global Value Numbering
  - Automatically done based on node equality

A Simple Optimization Phase

```
public class LockEliminationPhase extends Phase {

  @Override
  protected void run(StructuredGraph graph) {
    for (MonitorExitNode node : graph.getNodes(MonitorExitNode.class)) {
      FixedNode next = node.next();
      if (next instanceof MonitorEnterNode) {
        MonitorEnterNode monitorEnterNode = (MonitorEnterNode) next;
        if (monitorEnterNode.object() == node.object()) {
          GraphUtil.removeFixedWithUnusedInputs(monitorEnterNode);
          GraphUtil.removeFixedWithUnusedInputs(node);
        }
      }
    }
  }
}
```

Eliminate unnecessary release-reacquire of a monitor when no instructions are between

Iterate all nodes of a certain class

Modify the graph

Type System (Stamps)

- Every node has a `Stamp` that describes the possible values of the node
  - The kind of the value (object, integer, float)
  - But with additional details if available
  - Stamps form a lattice with `meet` (= union) and `join` (= intersection) operations

- `ObjectStamp`
  - Declared type: the node produces a value of this type, or any subclass
  - Exact type: the node produces a value of this type (exactly, not a subclass)
  - Value is never null (or always null)
- `IntegerStamp`
  - Number of bits used
  - Minimum and maximum value
  - Bits that are always set, bits that are never set
- `FloatStamp`

Speculative Optimizations

Motivating Example for Speculative Optimizations

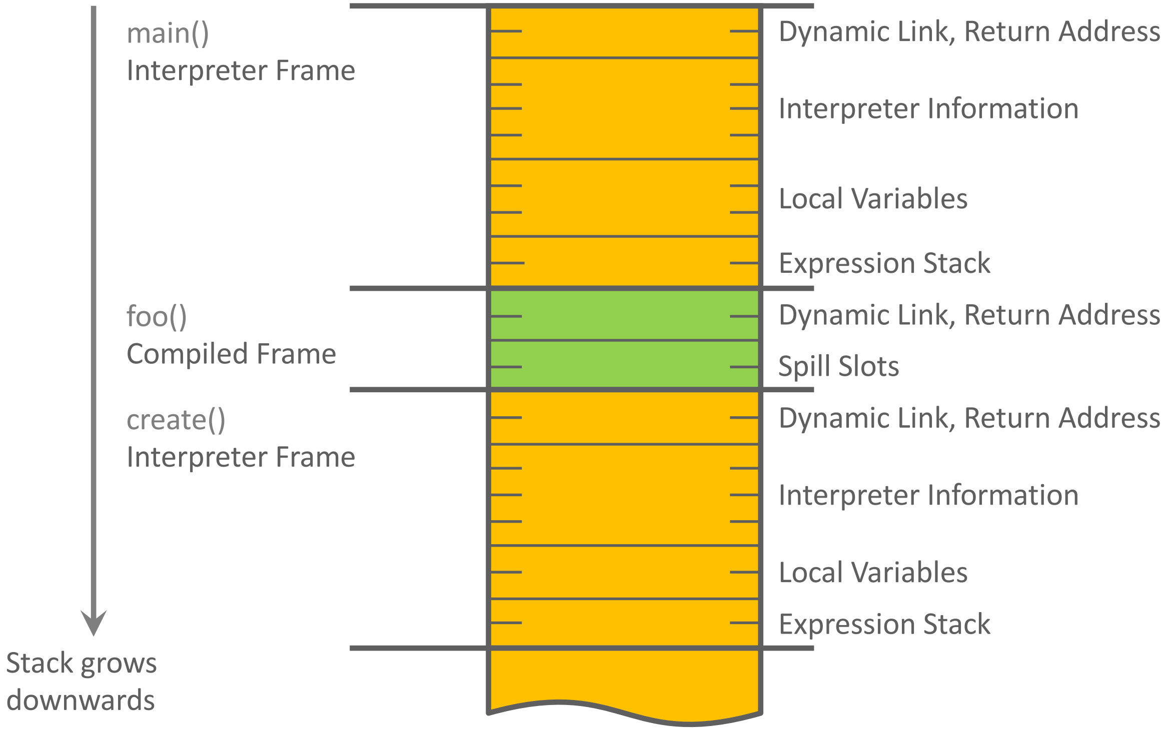
- Inlining of virtual methods
  - Most methods in Java are dynamically bound
  - Class Hierarchy Analysis
  - Inline when only one suitable method exists
- Compilation of foo() when only A loaded
  - Method getX() is inlined
  - Same machine code as direct field access
  - No dynamic type check
- Later loading of class B
  - Discard machine code of foo()
  - Recompile later without inlining
- Deoptimization
  - Switch to interpreter in the middle of foo()
  - Reconstruct interpreter stack frames
  - Expensive, but rare situation
  - Most classes already loaded at first compile

```
void foo() {
  A a = create();
  a.getX();
}
```

```
class A {
  int x;

  int getX() {
    return x;
  }
}
```

```
class B extends A {
  int getX() {
    return ...
  }
}
```

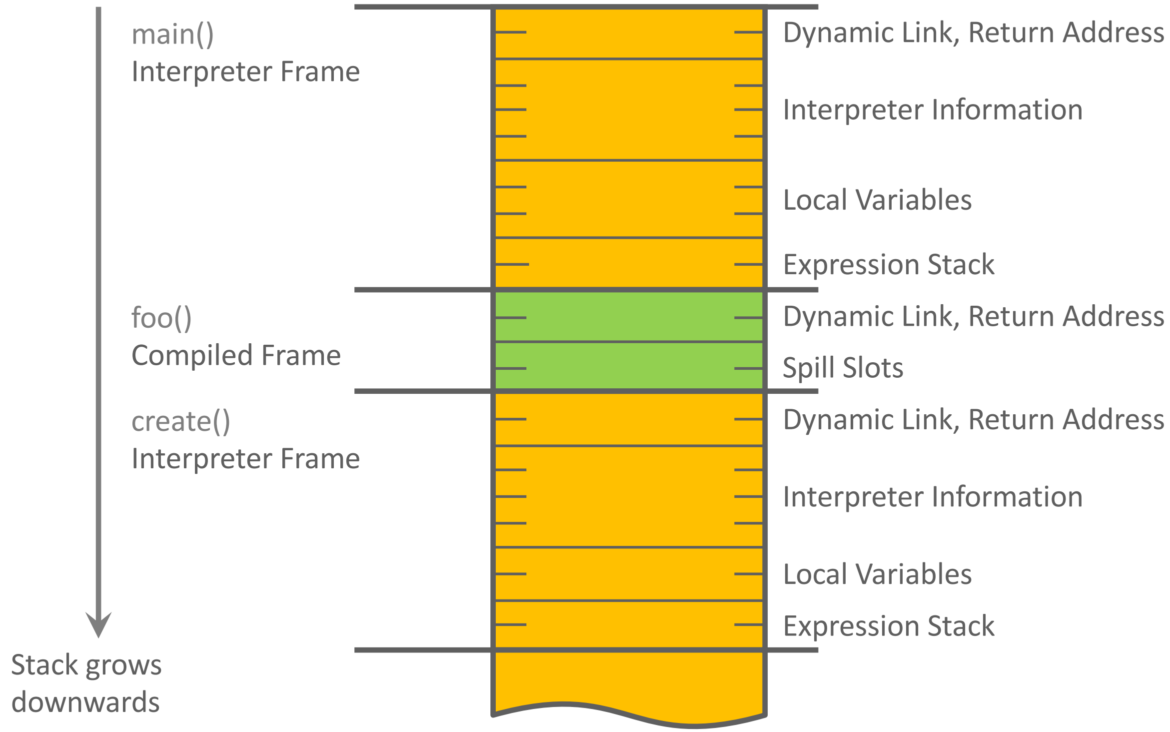
Deoptimization



Machine code for foo():

```
enter
call create
move [eax + 8] -> esi
leave
return
```

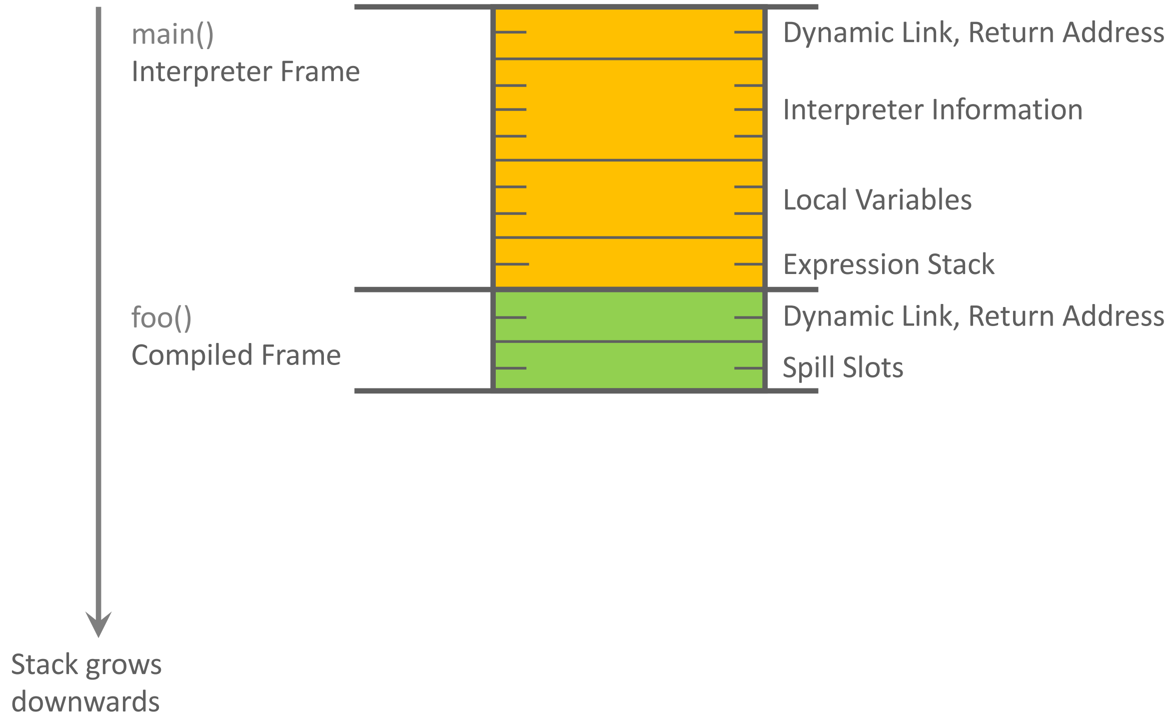
Deoptimization



Machine code for foo():

```
jump Interpreter
call create
call Deoptimization
leave
return
```

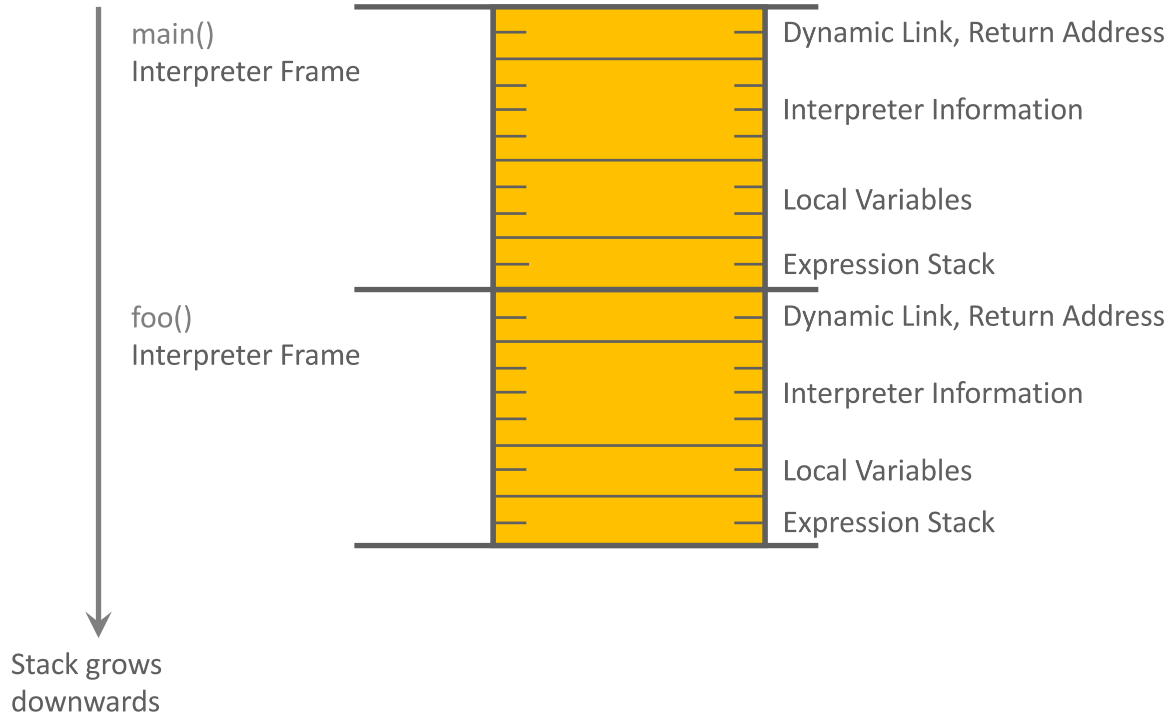
Deoptimization



Machine code for foo():

```
jump Interpreter
call create
call Deoptimization
leave
return
```

Deoptimization



Machine code for foo():

```
jump Interpreter
call create
call Deoptimization
leave
return
```

Example: Speculative Optimization

Java source code:

```
int f1;
int f2;

void speculativeOptimization(boolean flag) {
  f1 = 41;
  if (flag) {
    f2 = 42;
    return;
  }
  f2 = 43;
}
```

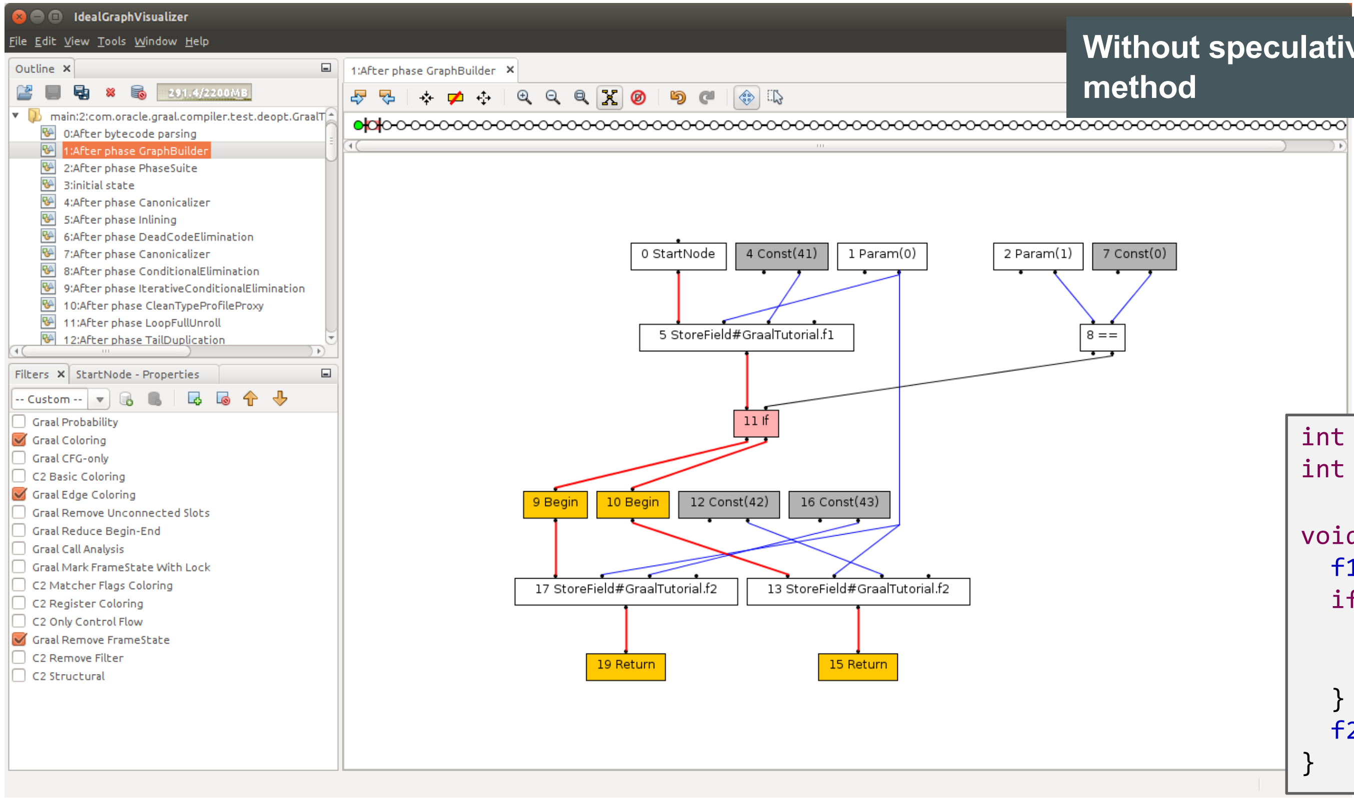
Assumption: method `speculativeOptimization` is always called with parameter `flag` set to `false`

Command line to run example:

```
./mx.sh igv &
./mx.sh unittest -G:Dump= -G:MethodFilter=GraalTutorial.speculativeOptimization GraalTutorial#testSpeculativeOptimization
```

The test case dumps two graphs: first with speculation, then without speculation

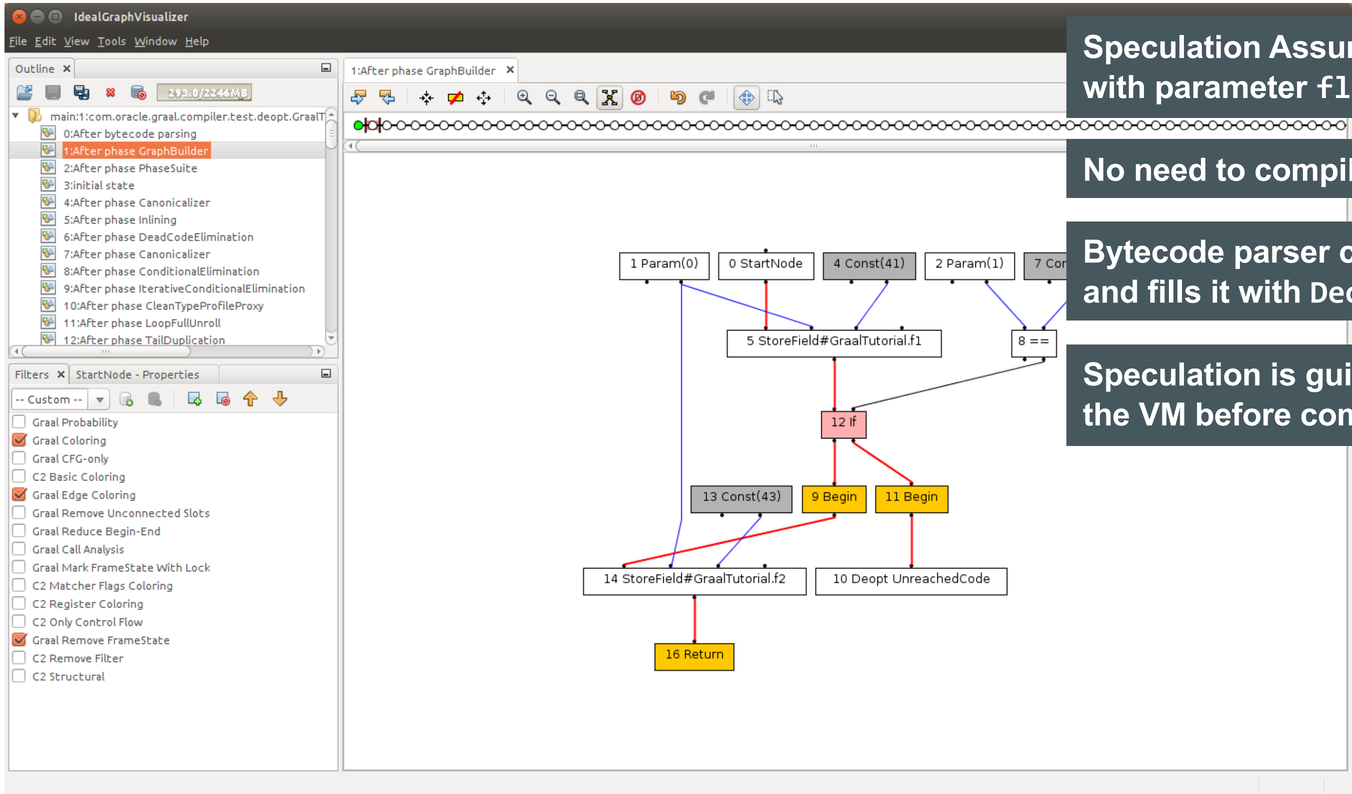
After Parsing without Speculation



Without speculative optimizations: graph covers the whole method

```java
int f1;
int f2;

void speculativeOptimization(boolean flag) {
  f1 = 41;
  if (flag) {
    f2 = 42;
    return;
  }
  f2 = 43;
}
```

After Parsing with Speculation



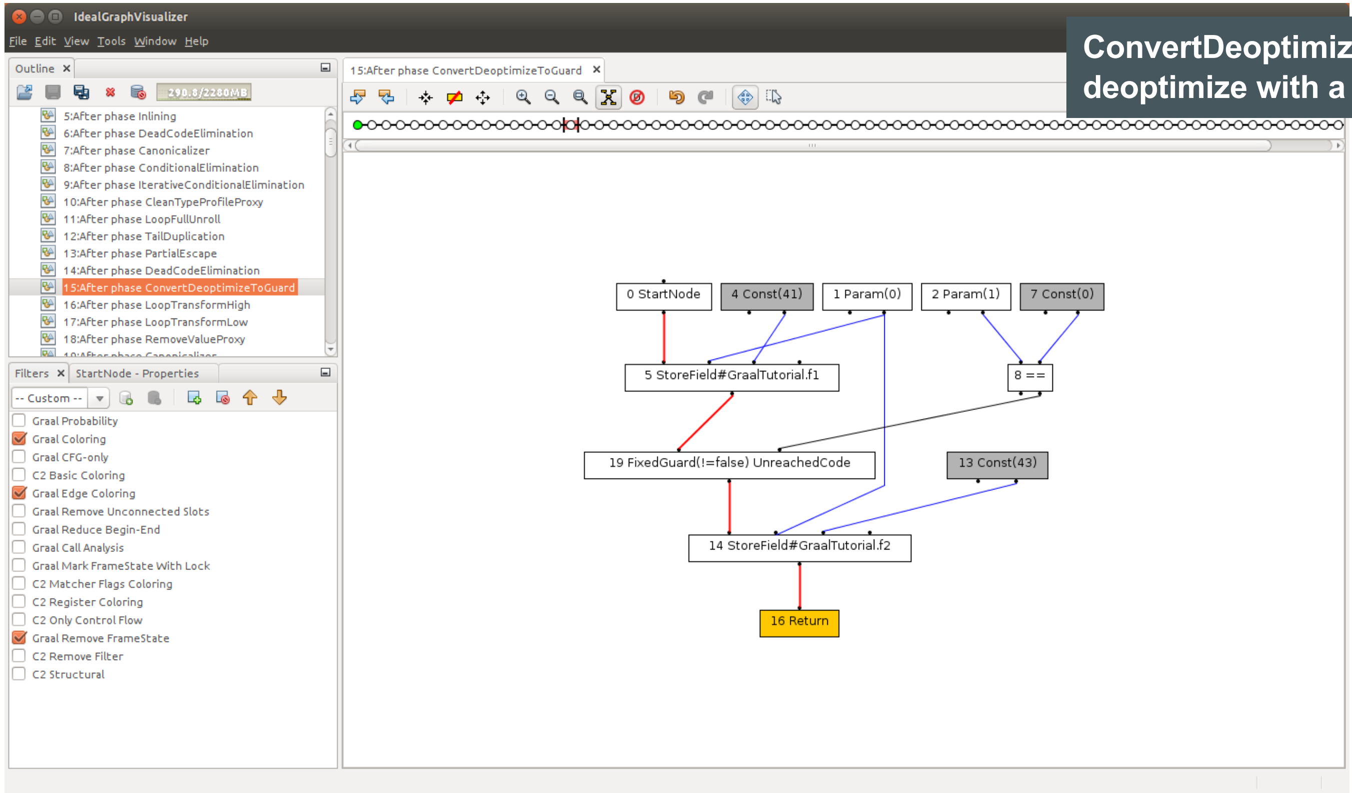
Speculation Assumption: method `test` is always called with parameter `flag` set to `false`

No need to compile the code inside the `if` block

Bytecode parser creates the `if` block, but stops parsing and fills it with `DeoptimizeNode`

Speculation is guided by profiling information collected by the VM before compilation

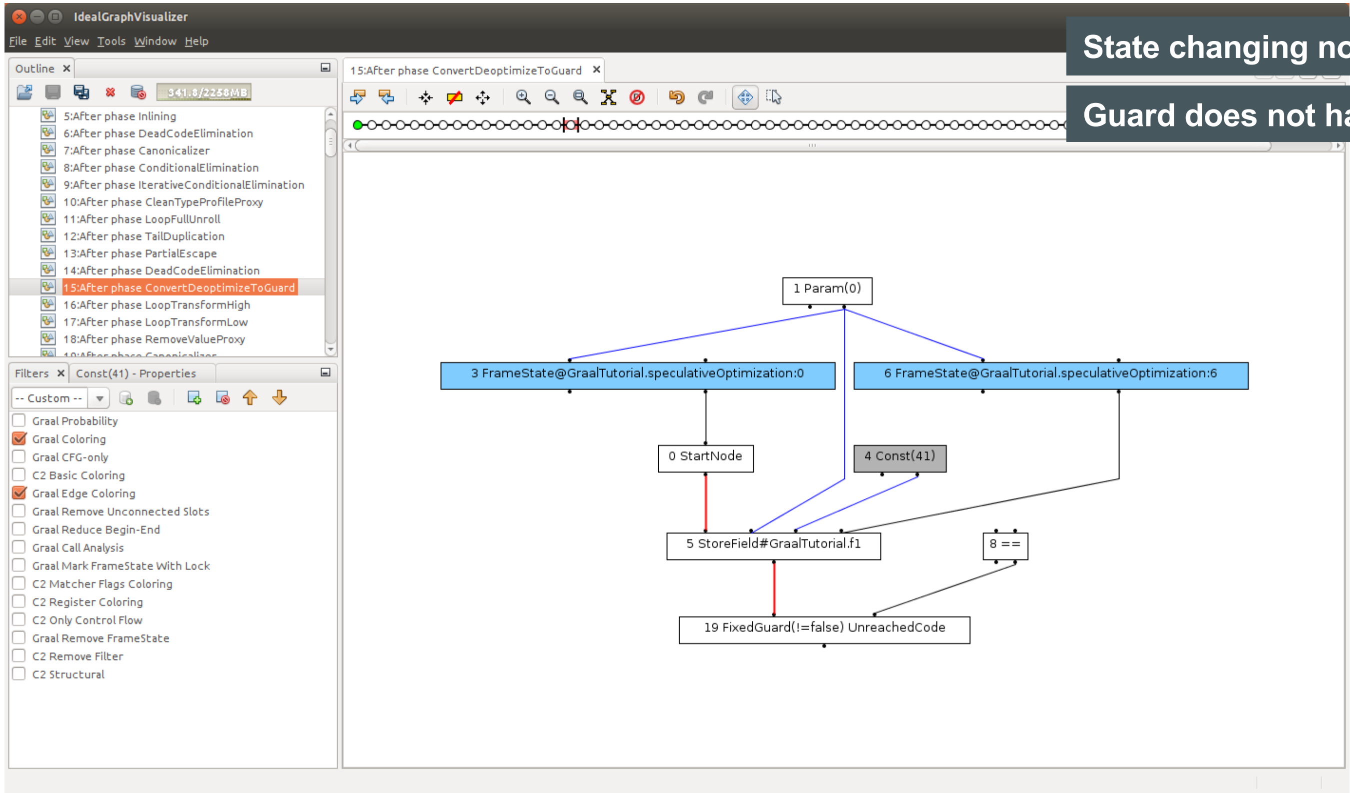
After Converting Deoptimize to Fixed Guard



ConvertDeoptimizeToGuardPhase replaces the if-deoptimize with a single FixedGuardNode

ORACLE

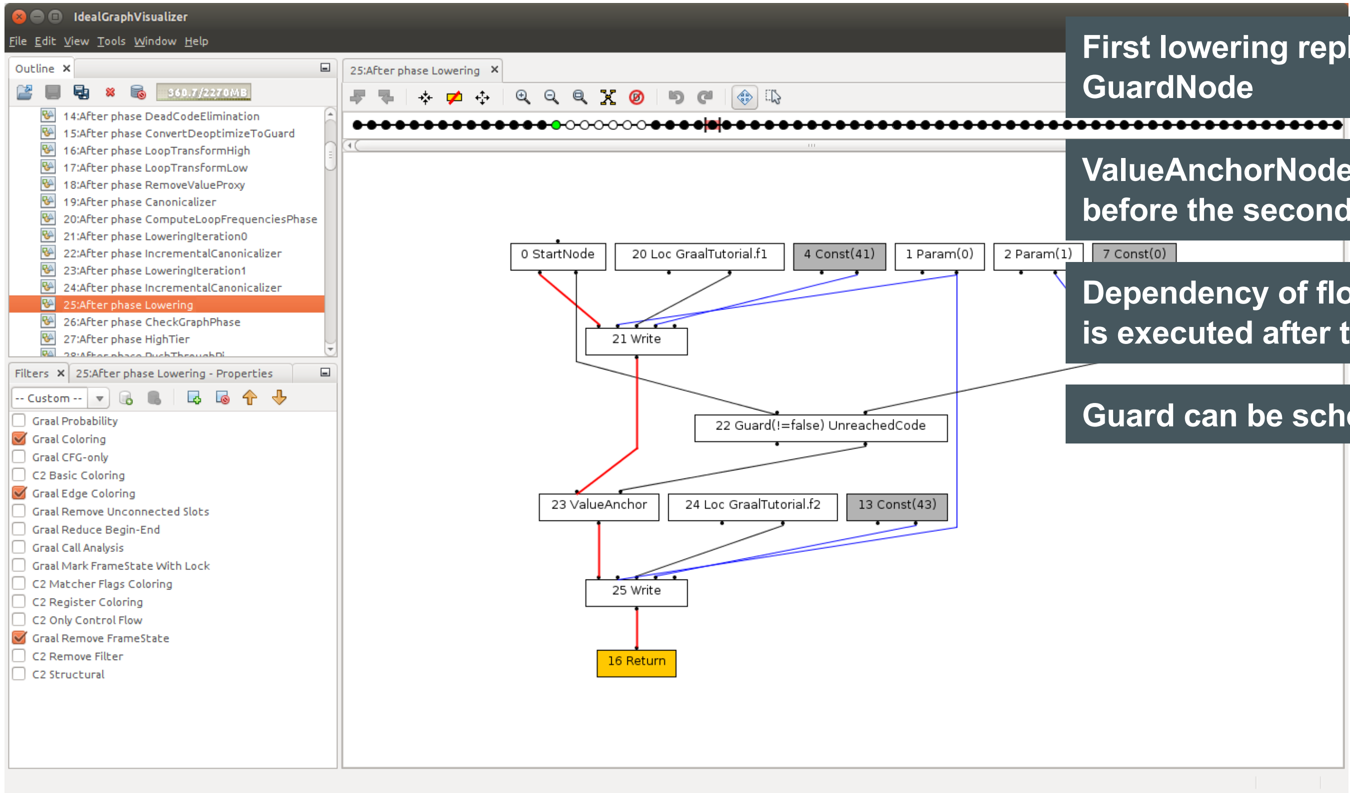
Frame states after Parsing



State changing nodes have a FrameState

Guard does not have a FrameState

After Lowering: Guard is Floating



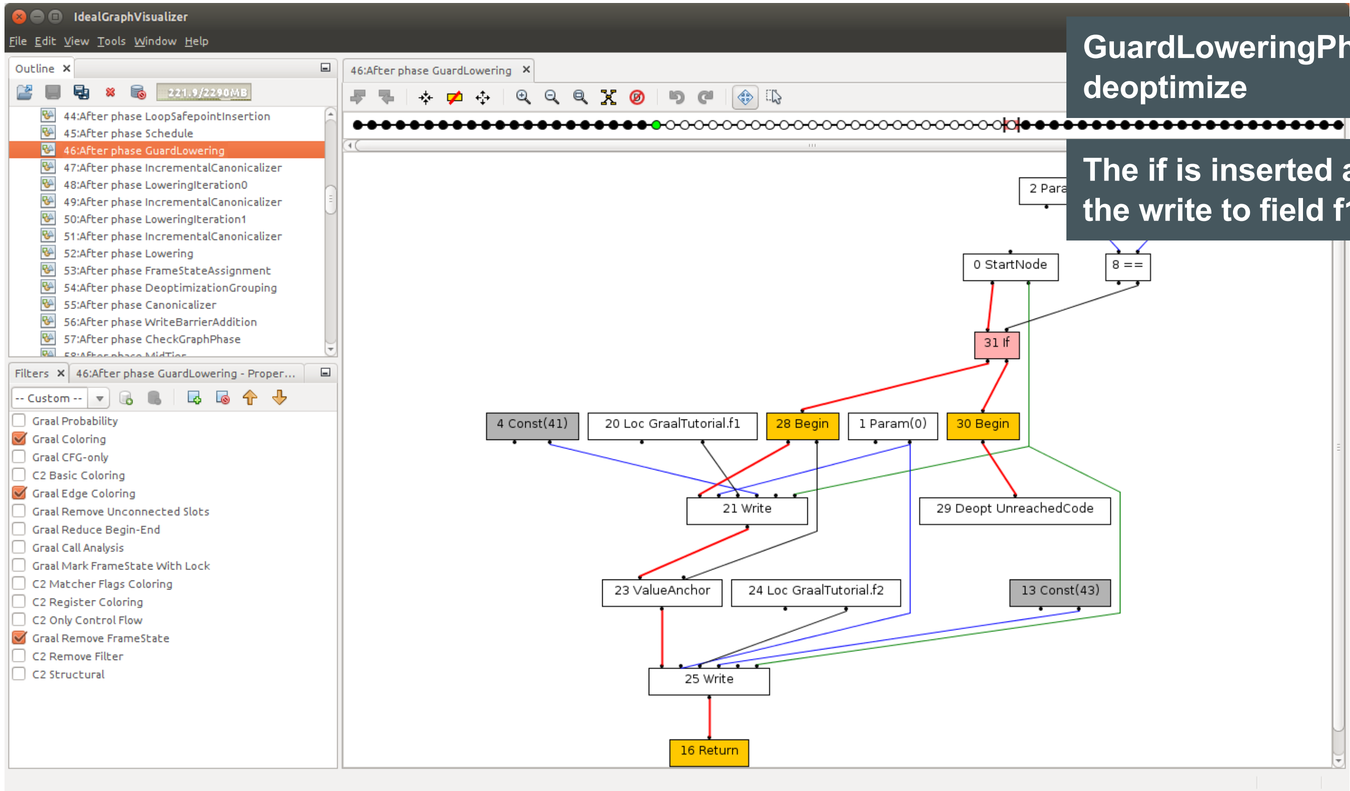
First lowering replaces the FixedGuardNode with a floating GuardNode

ValueAnchorNode ensures the floating guard is executed before the second write

Dependency of floating guard on StartNode ensures guard is executed after the method start

Guard can be scheduled within these constraints

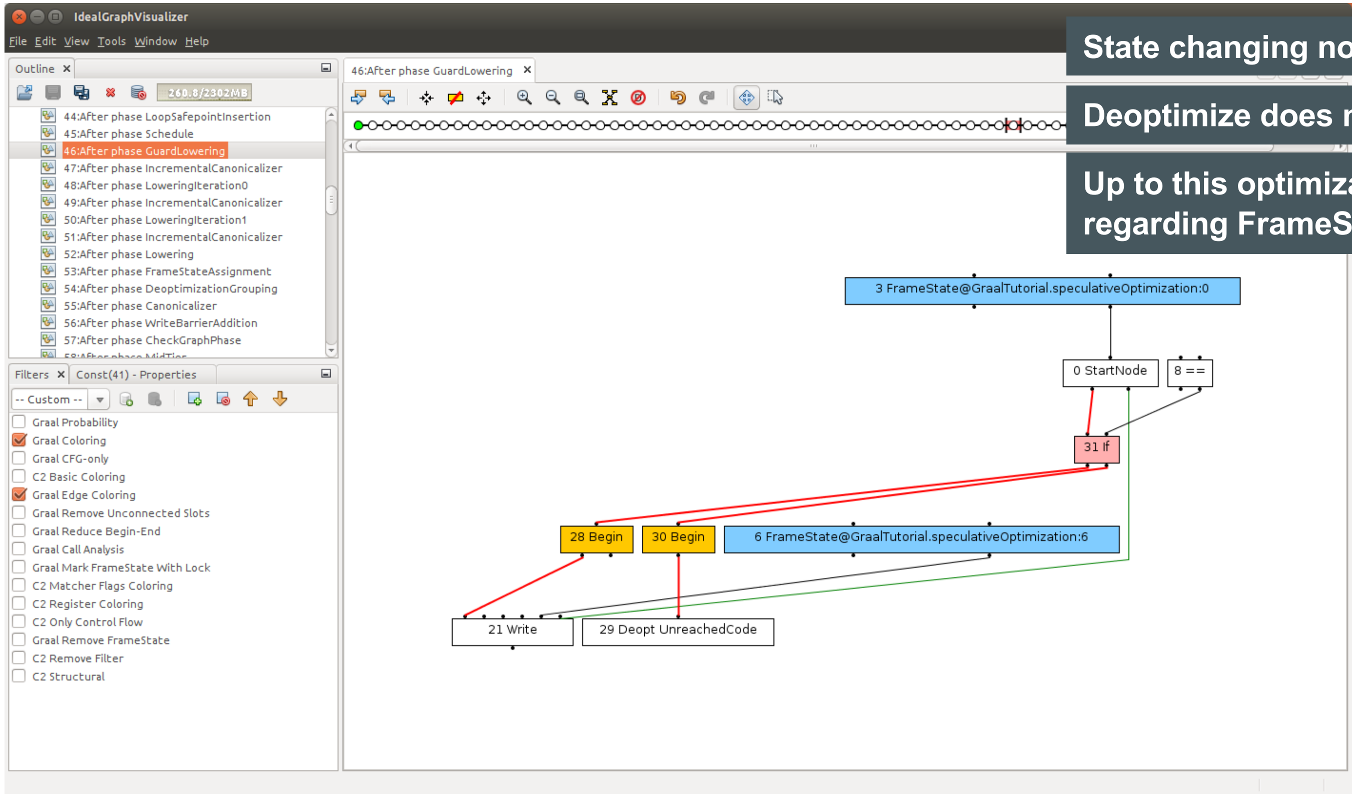
After Replacing Guard with If-Deoptimize



GuardLoweringPhase replaces GuardNode with if-deoptimize

The if is inserted at the best (earliest) position – it is before the write to field f1

Frame States are Still Unchanged



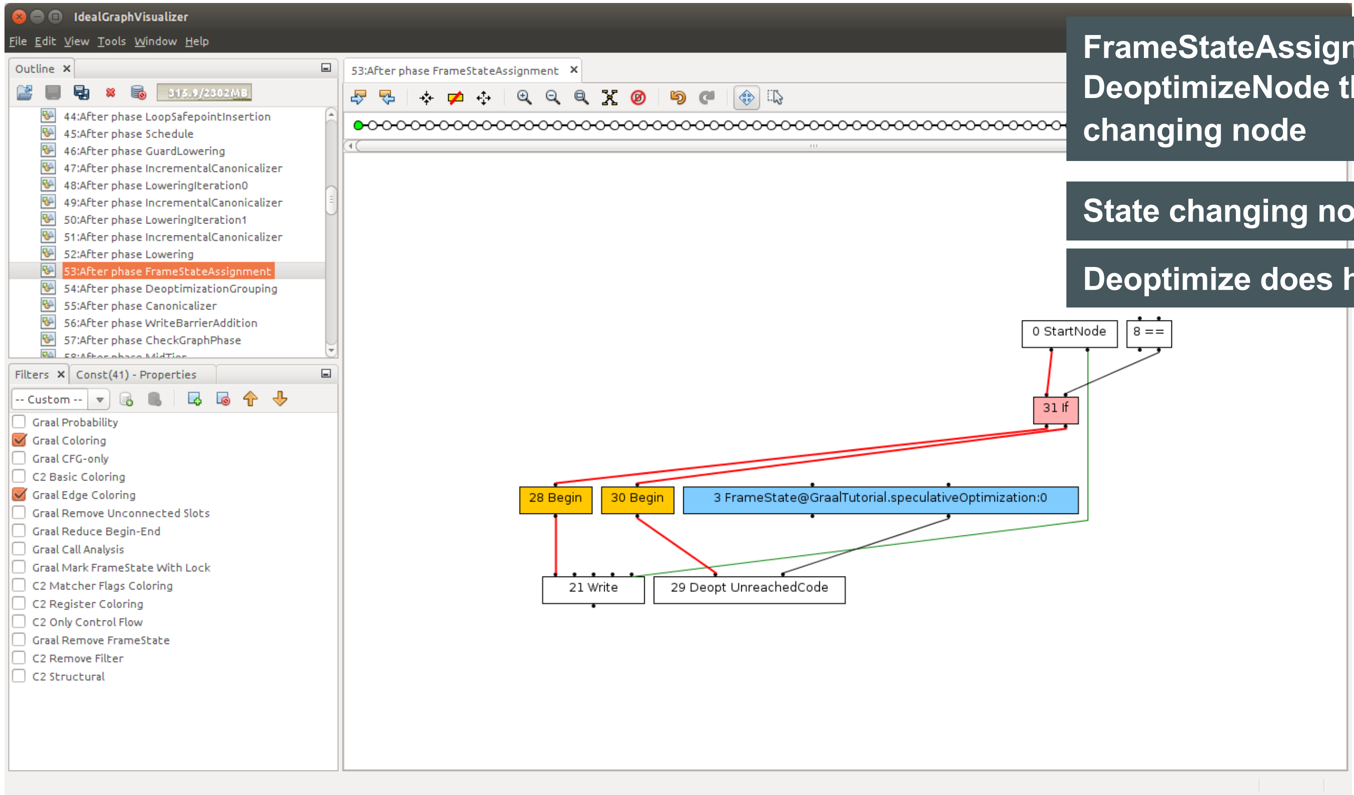
State changing nodes have a FrameState

Deoptimize does not have a FrameState

Up to this optimization stage, nothing has changed regarding FrameState nodes

ORACLE

After FrameStateAssignmentPhase

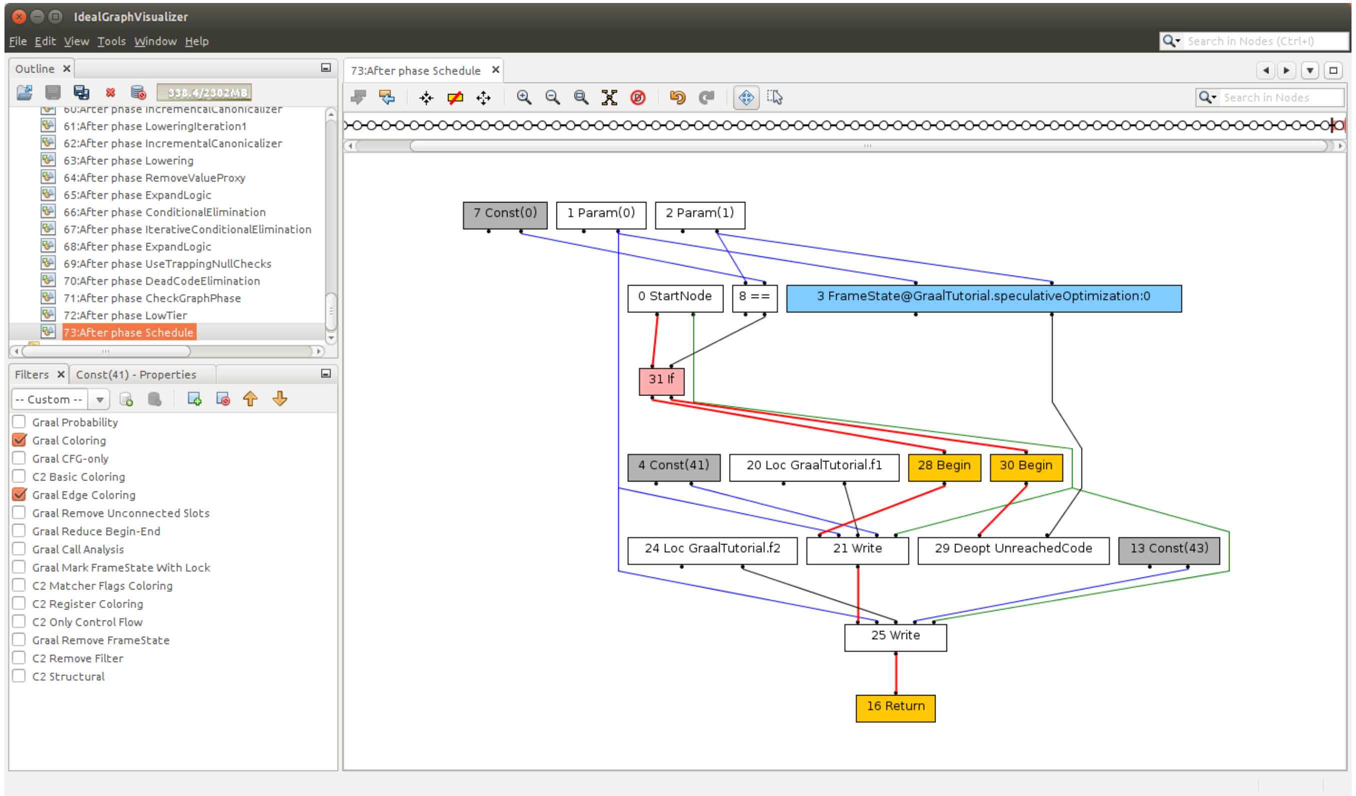


FrameStateAssignmentPhase assigns every DeoptimizeNode the FrameState of the preceding state changing node

State changing nodes do not have a FrameState

Deoptimize does have a FrameState

Final Graph After Optimizations



Frame States: Two Stages of Compilation

	First Stage: Guard Optimizations	Second Stage: Side-effects Optimizations
`FrameState` is on …	… nodes with side effects	… nodes that deoptimize
Nodes with side effects …	… cannot be moved within the graph	… can be moved
Nodes that deoptimize …	… can be moved within the graph	… cannot be moved
	New guards can be introduced anywhere at any time. Redundant guards can be eliminated. Most optimizations are performed in this stage.	Nodes with side effects can be reordered or combined.
`StructuredGraph.guardsStage =`	`GuardsStage.FLOATING_GUARDS`	`GuardsStage.AFTER_FSA`
Graph is in this stage …	… before `GuardLoweringPhase`	… after `FrameStateAssignmentPhase`

Implementation note: Between `GuardLoweringPhase` and `FrameStateAssignmentPhase`, the graph is in stage `GuardsStage.FIXED_DEOPTS`. This stage has no benefit for optimization, because it has the restrictions of both major stages.

Optimizations on Floating Guards

- Redundant guards are eliminated
  - Automatically done by global value numbering
  - Example: multiple bounds checks on the same array

- Guards are moved out of loops
  - Automatically done by scheduling
  - GuardLoweringPhase assigns every guard a dependency on the reverse postdominator of the original fixed location
    - The block whose execution guarantees that the original fixed location will be reached too
  - For guards in loops (but not within a if inside the loop), this is a block before the loop

- Speculative optimizations can move guards further up
  - This needs a feedback cycle with the interpreter: if the guard actually triggers deoptimization, subsequent recompilation must not move the guard again

ORACLE

Graal API

Graal API Interfaces

- Interfaces for everything coming from a .class file
  - `JavaType`, `JavaMethod`, `JavaField`, `ConstantPool`, `Signature`, …
- Provider interfaces
  - `MetaAccessProvider`, `CodeCacheProvider`, `ConstantReflectionProvider`, …
- VM implements the interfaces, Graal uses the interfaces

- `CompilationResult` is produced by Graal
  - Machine code in `byte[]` array
  - Pointer map information for garbage collection
  - Information about local variables for deoptimization
  - Information about speculations performed during compilation

Dynamic Class Loading

- From the Java specification: Classes are loaded and initialized as late as possible
  - Code that is never executed can reference a non-existing class, method, or field
  - Invoking a method does not make the whole method executed
  - Result: Even a frequently executed (= compiled) method can have parts that reference non-existing elements
  - The compiler must not trigger class loading or initialization, and must not throw linker errors
- Graal API distinguishes between unresolved and resolved elements
  - Interfaces for unresolved elements: `JavaType`, `JavaMethod`, `JavaField`
    - Only basic information: name, field kind, method signature
  - Interfaces for resolved elements: `ResolvedJavaType`, `ResolvedJavaMethod`, `ResolvedJavaField`
    - All the information that Java reflection gives you, and more
- Graal as a JIT compiler does not trigger class loading
  - Replace accesses to unresolved elements with deoptimization, let interpreter then do the loading and linking
- Graal as a static analysis framework can trigger class loading

Important Provider Interfaces

```
public interface MetaAccessProvider {
  ResolvedJavaType lookupJavaType(Class<?> clazz);
  ResolvedJavaMethod lookupJavaMethod(Executable reflectionMethod);
  ResolvedJavaField lookupJavaField(Field reflectionField);
  ...
}
```

Convert Java reflection objects to Graal API

```
public interface ConstantReflectionProvider {
  Boolean constantEquals(Constant x, Constant y);
  Integer readArrayLength(JavaConstant array);
  ...
}
```

Look into constants – note that the VM can deny the request, maybe it does not even have the information

It breaks the compiler-VM separation to get the raw object encapsulated in a Constant – so there is no method for it

```
public interface CodeCacheProvider {
  InstalledCode addMethod(ResolvedJavaMethod method, CompilationResult compResult,
          SpeculationLog speculationLog, InstalledCode predefinedInstalledCode);
  InstalledCode setDefaultMethod(ResolvedJavaMethod method, CompilationResult compResult);
  TargetDescription getTarget();
  ...
}
```

Install compiled code into the VM

Example: Print Bytecodes of a Method

```
/* Entry point object to the Graal API from the hosting VM. */
RuntimeProvider runtimeProvider = Graal.getRequiredCapability(RuntimeProvider.class);

/* The default backend (architecture, VM configuration) that the hosting VM is running on. */
Backend backend = runtimeProvider.getHostBackend();

/* Access to all of the Graal API providers, as implemented by the hosting VM. */
Providers providers = backend.getProviders();

/* The provider that allows converting reflection objects to Graal API. */
MetaAccessProvider metaAccess = providers.getMetaAccess();

Method reflectionMethod = ...
ResolvedJavaMethod method = metaAccess.lookupJavaMethod(reflectionMethod);

/* ResolvedJavaMethod provides all information that you want about a method, for example, the bytecodes. */
byte[] bytecodes = method.getCode();

/* BytecodeDisassembler shows you how to iterate bytecodes, how to access type information, and more. */
System.out.println(new BytecodeDisassembler().disassemble(method));
```

Command line to run example:

```
./mx.sh unittest GraalTutorial#testPrintBytecodes
```

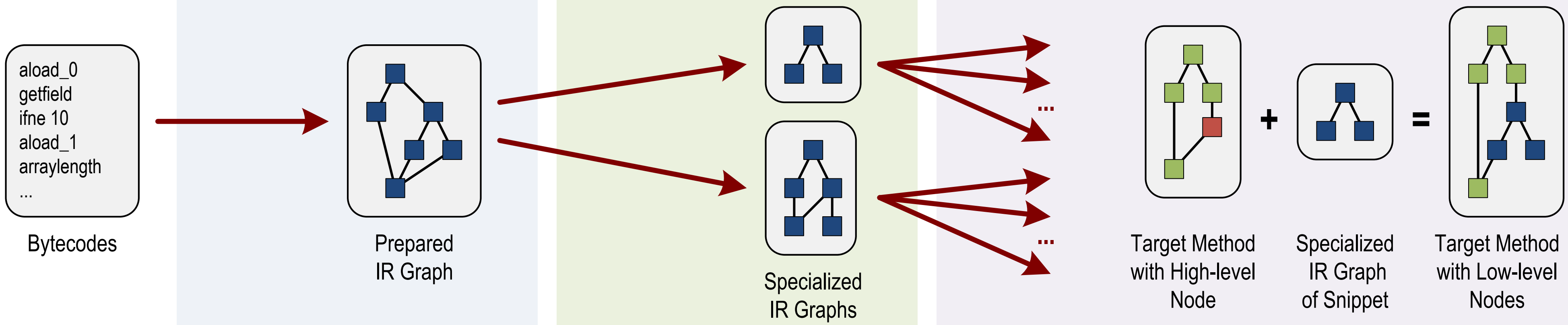
Snippets

The Lowering Problem

- How do you express the low-level semantics of a high-level operation?
- Manually building low-level IR graphs
  - Tedious and error prone
- Manually generating machine code
  - Tedious and error prone
  - Probably too low level (no more compiler optimizations possible after lowering)

- Solution: Snippets
  - Express the semantics of high-level Java operations in low-level Java code
    - `Word` type representing a machine word allows raw memory access
  - Simplistic view: replace a high-level node with an inlined method
  - To make it work in practice, a few more things are necessary

Snippet Lifecycle



	Preparation	Specialization	Instantiation
Frequency:	Once	Few Times	Many Times
Steps:	Java Bytecode Parsing Exhaustive Method Inlining Node Intrinsification Constant Folding, Canonicalization	Graph Duplication Constant Parameter Replacement Node Intrinsification Constant Folding, Canonicalization	Graph Duplication Graph Inlining in Target Method Constant Folding, Canonicalization

Snippet Example: instanceOf with Profiling Information

```
@Snippet
static Object instanceofWithProfile(Object object,
      @ConstantParameter boolean nullSeen,
      @VarargsParameter Word[] profiledHubs,
      @VarargsParameter boolean[] hubIsPositive) {

  if (probability(NotFrequent, object == null)) {
    if (!nullSeen) {
      deoptimize(OptimizedTypeCheckViolated);
      throw shouldNotReachHere();
    }
    isNullCounter.increment();
    return false;
  }
  Anchor afterNullCheck = anchor();
  Word objectHub = loadHub(object, afterNullCheck);

  explodeLoop();
  for (int i = 0; i < profiledHubs.length; i++) {
    if (profiledHubs[i].equal(objectHub)) {
      profileHitCounter.increment();
      return hubIsPositive[i];
    }
  }
  deoptimize(OptimizedTypeCheckViolated);
  throw shouldNotReachHere();
}
```

Constant folding during specialization

Loop unrolling during specialization

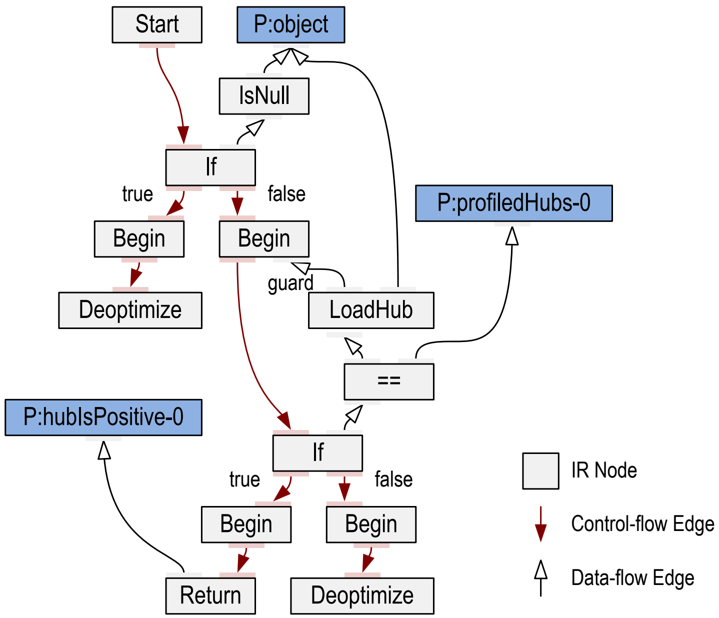
Node intrinsic

Debug / profiling code eliminated by constant folding and dead code elimination

Loop unrolling during specialization

Snippet Example: Specialization for One Type

```
@Snippet
static Object instanceofWithProfile(Object object,
      @ConstantParameter boolean nullSeen,
      @VarargsParameter Word[] profiledHubs,
      @VarargsParameter boolean[] hubIsPositive) {

  if (probability(NotFrequent, object == null)) {
    if (!nullSeen) {
      deoptimize(OptimizedTypeCheckViolated);
      throw shouldNotReachHere();
    }
    isNullCounter.increment();
    return false;
  }
  Anchor afterNullCheck = anchor();
  Word objectHub = loadHub(object, afterNullCheck);

  explodeLoop();
  for (int i = 0; i < profiledHubs.length; i++) {
    if (profiledHubs[i].equal(objectHub)) {
      profileHitCounter.increment();
      return hubIsPositive[i];
    }
  }
  deoptimize(OptimizedTypeCheckViolated);
  throw shouldNotReachHere();
}
```



Node Intrinsics

```
class LoadHubNode extends FloatingGuardedNode {

  @Input ValueNode object;

  LoadHubNode(ValueNode object, ValueNode guard) {
    super(guard);
    this.object = object;
  }
}

@NodeIntrinsic(LoadHubNode.class)
static native Word loadHub(Object object, Object guard);
```

Calling the node intrinsic reflectively instantiates the node using the matching constructor

```
class DeoptimizeNode extends ControlSinkNode {

  final Reason reason;

  DeoptimizeNode(Reason reason) {
    this.object = object;
  }
}

@NodeIntrinsic(DeoptimizeNode.class)
static native void deoptimize(
                       @ConstantNodeParameter Reason reason);
```

Constructor with non-Node parameter requires node intrinsic parameter to be a constant during snippet specialization

Snippet Instantiation

```
SnippetInfo instanceofWithProfile = snippet(InstanceOfSnippets.class, "instanceofWithProfile");

void lower(InstanceOfNode node) {
  ValueNode object = node.getObject();
  JavaTypeProfile profile = node.getProfile();

  if (profile.totalProbability() > threshold) {
    int numTypes = profile.getNumTypes();
    Word[] profiledHubs = new Word[numTypes];
    boolean hubIsPositive = new boolean[numTypes];
    for (int i = 0; i < numTypes; i++) {
      profiledHubs[i] = profile.getType(i).getHub();
      hubIsPositive[i] = profile.isPositive(i);
    }

    Args args = new Args(instanceofWithProfile);
    args.add(object);
    args.addConst(profile.getNullSeen());
    args.addVarargs(profiledHubs);
    args.addVarargs(hubIsPositive);

    SnippetTemplate s = template(args);
    s.instantiate(args, node);

  } else {
    // Use a different snippet.
  }
}
```

Node argument: formal parameter of snippet is replaced with this node

Constant argument for snippet specialization

Snippet preparation and specialization

Snippet instantiation

Example in IGV

- The previous slides are slightly simplified
  - In reality the snippet graph is a bit more complex
  - But the end result is the same

The snippets for lowering of instanceOf are in class `InstanceOfSnippets`

Java source code:

```
static class A { }
static class B extends A { }

static int instanceOfUsage(Object obj) {
  if (obj instanceof A) {
    return 42;
  } else {
    return 0;
  }
}
```

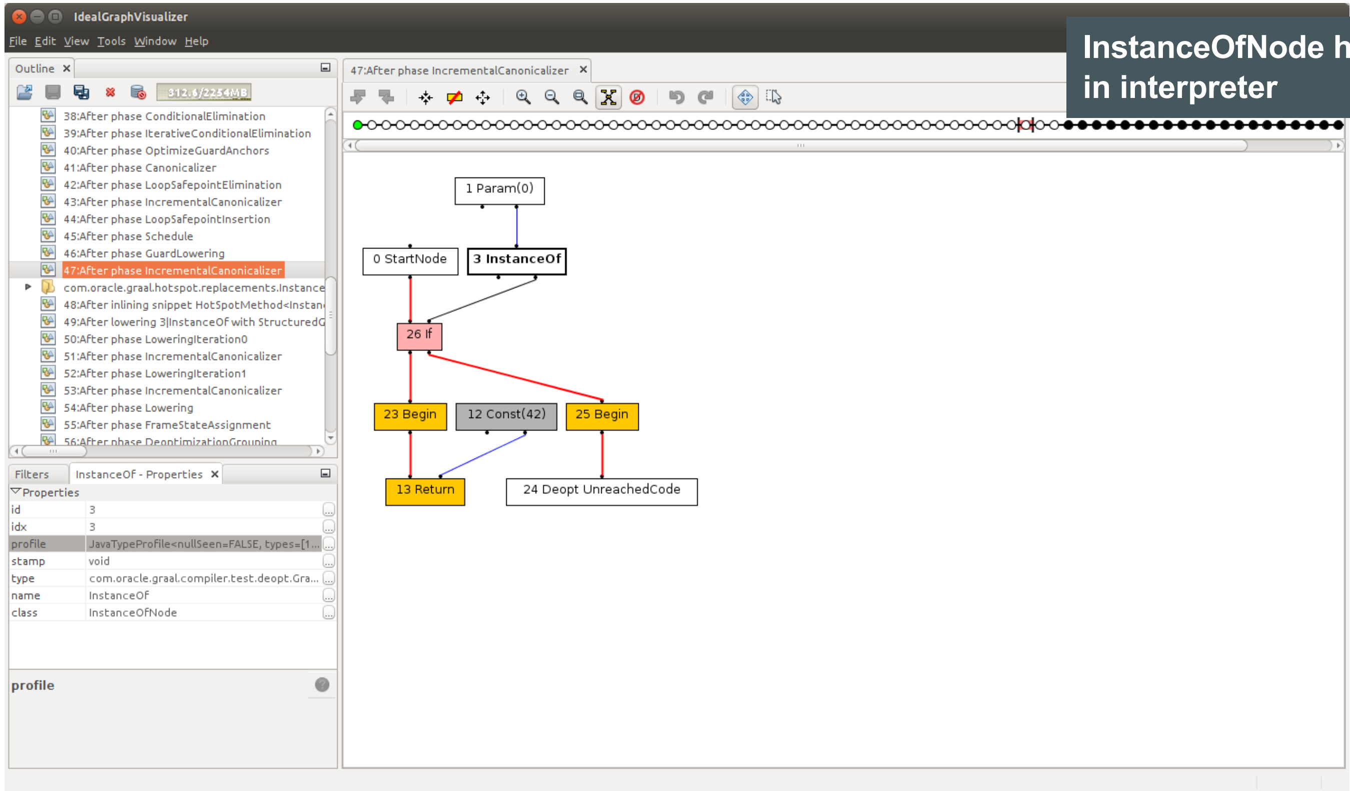
Assumption: method `instanceOfUsage` is always called with parameter `obj` having class `A`

Command line to run example:

```
./mx.sh igv &
./mx.sh unittest -G:Dump= -G:MethodFilter=GraalTutorial.instanceOfUsage GraalTutorial#testInstanceOfUsage
```

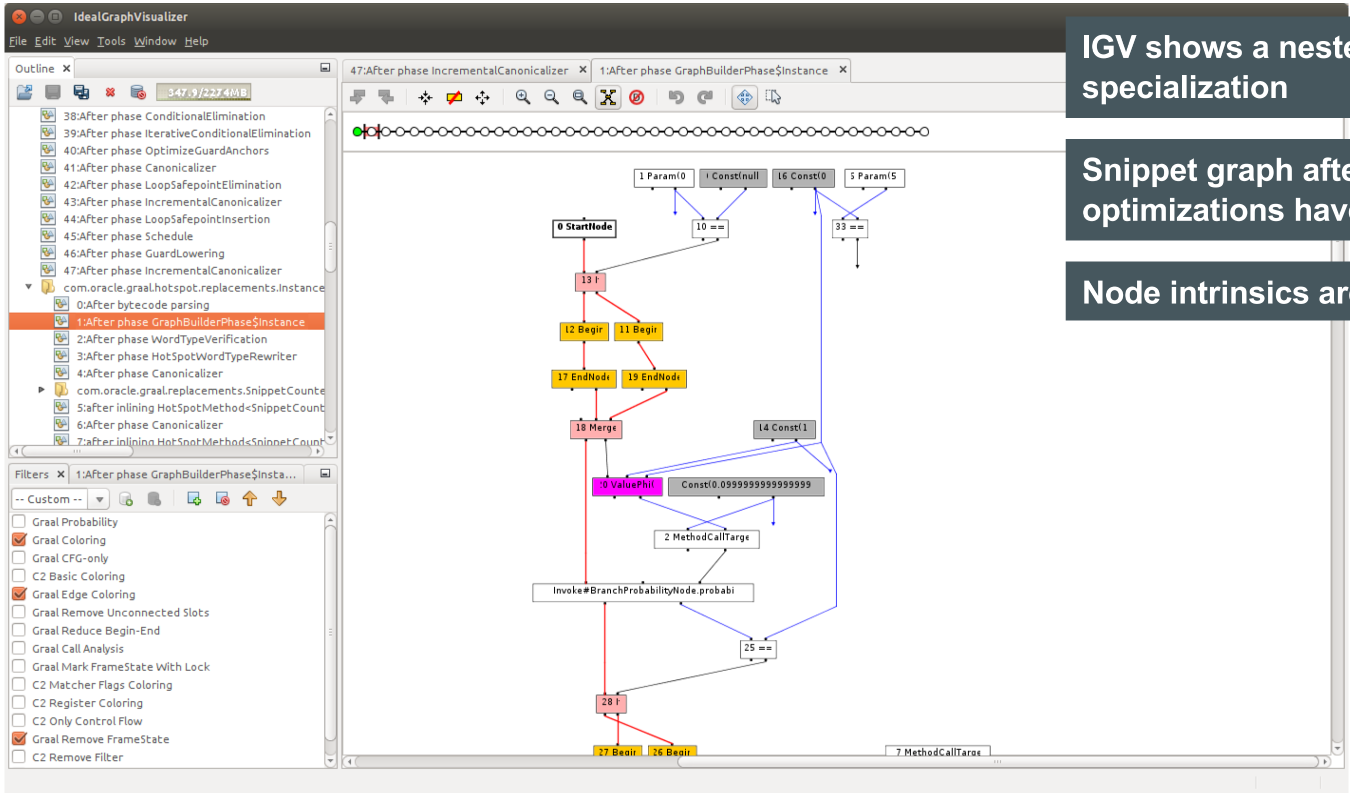
ORACLE

Method Before Lowering



InstanceOfNode has profiling information: only type A seen in interpreter

Snippet After Parsing

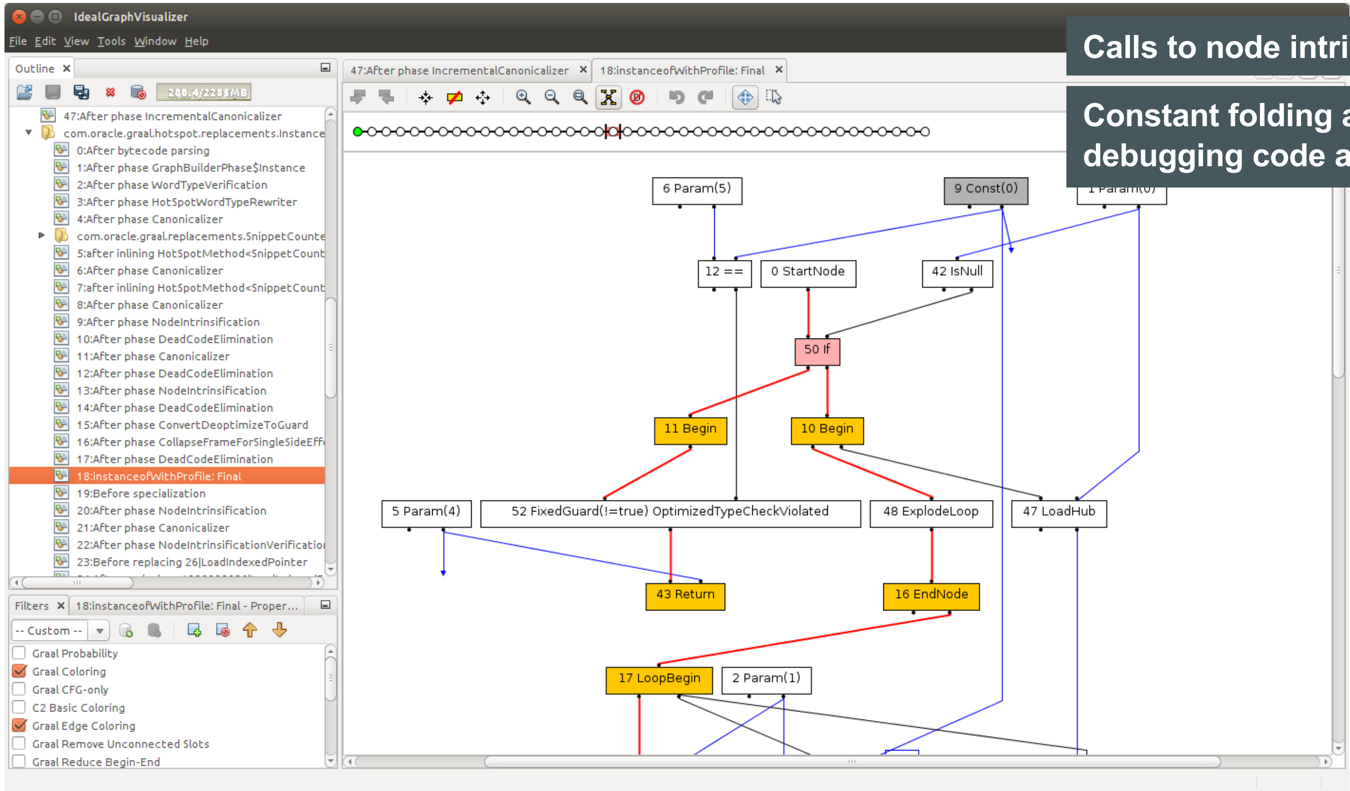


IGV shows a nested graph for snippet preparation and specialization

Snippet graph after bytecode parsing is big, because no optimizations have been performed yet

Node intrinsics are still method calls

Snippet After Preparation

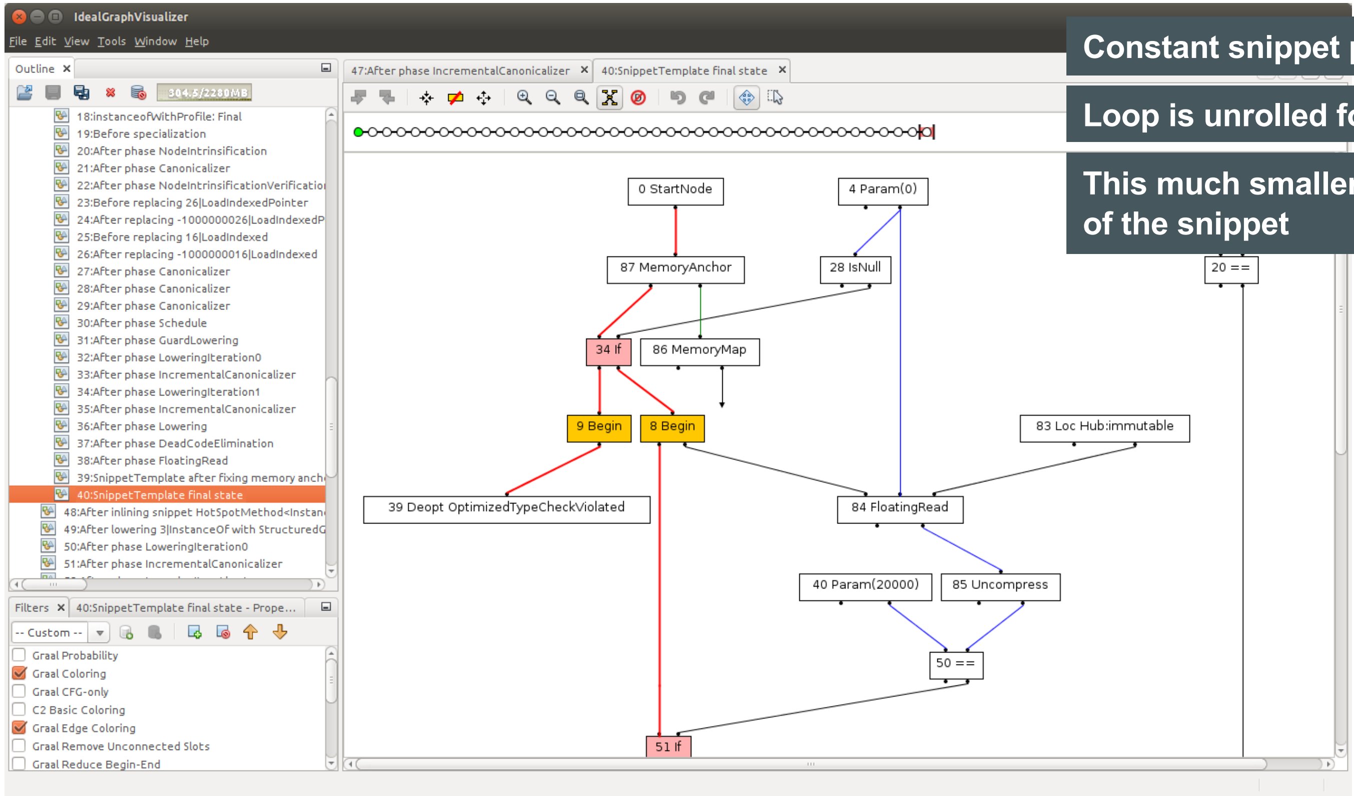


Calls to node intrinsics are replaced with actual nodes

Constant folding and dead code elimination removed debugging code and counters

ORACLE

Snippet After Specialization

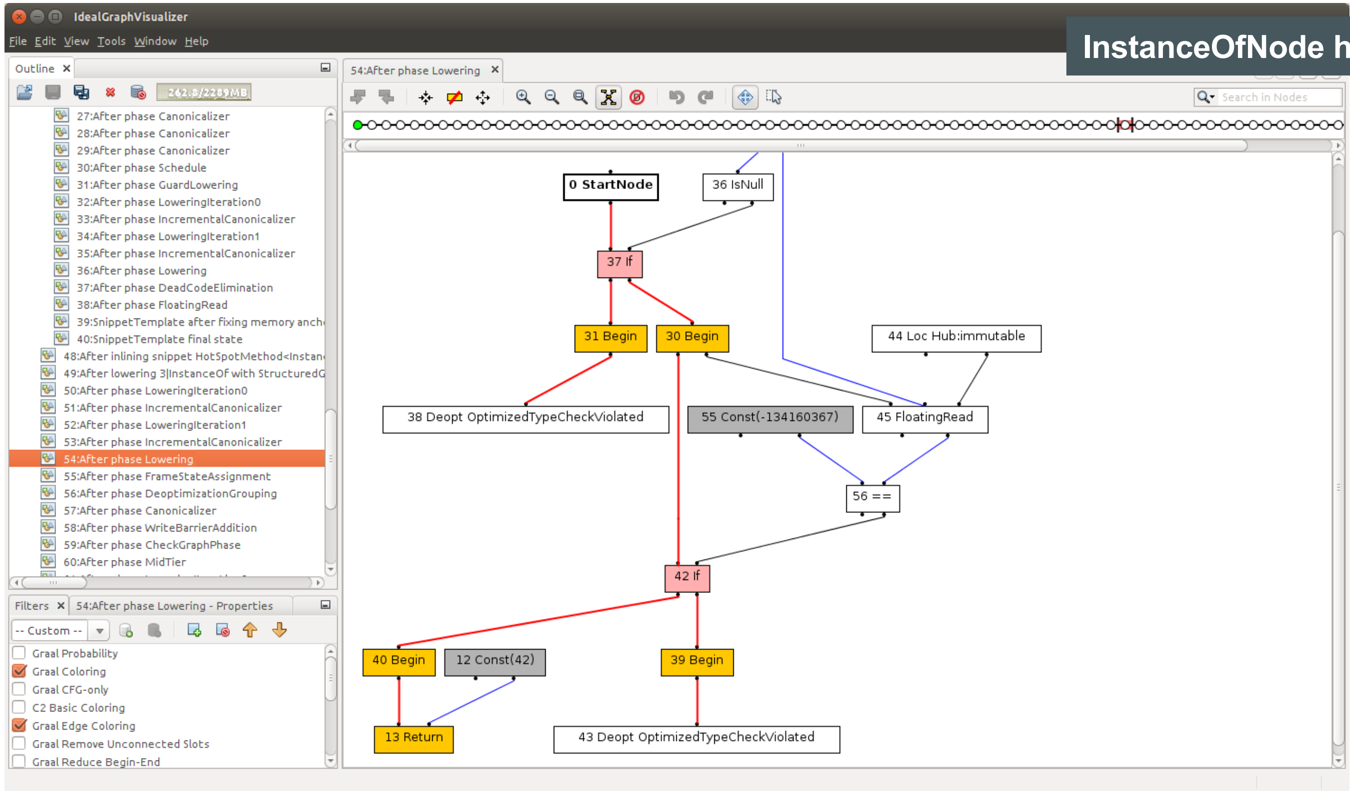


Constant snippet parameter is constant folded

Loop is unrolled for length 1

This much smaller graph is cached for future instantiations of the snippet

Method After Lowering



InstanceOfNode has been replaced with snippet graph

ORACLE

Compiler Intrinsics

Compiler Intrinsics

- Called “method substitution” in Graal
  - A lot mechanism and infrastructure shared with snippets
- Use cases
  - Use a special hardware instruction instead of calling a Java method
  - Replace a runtime call into the VM with low-level Java code
- Implementation steps
  - Define a node for the intrinsic functionality
  - Define a method substitution for the Java method that should be intrinsified
    - Use a node intrinsic to create your node
  - Define a LIR instruction for your functionality
  - Generate this LIR instruction in the LIRLowerable.generate() method of your node
  - Generate machine code in your LIRInstruction.emitCode() method

ORACLE

Example: Intrinsification of Math.sin()

Java source code:

```
static double intrinsicUsage(double val) {
  return Math.sin(val);
}
```

Java implementation of Math.sin() calls native code via JNI

x86 provides an FPU instruction: fsin

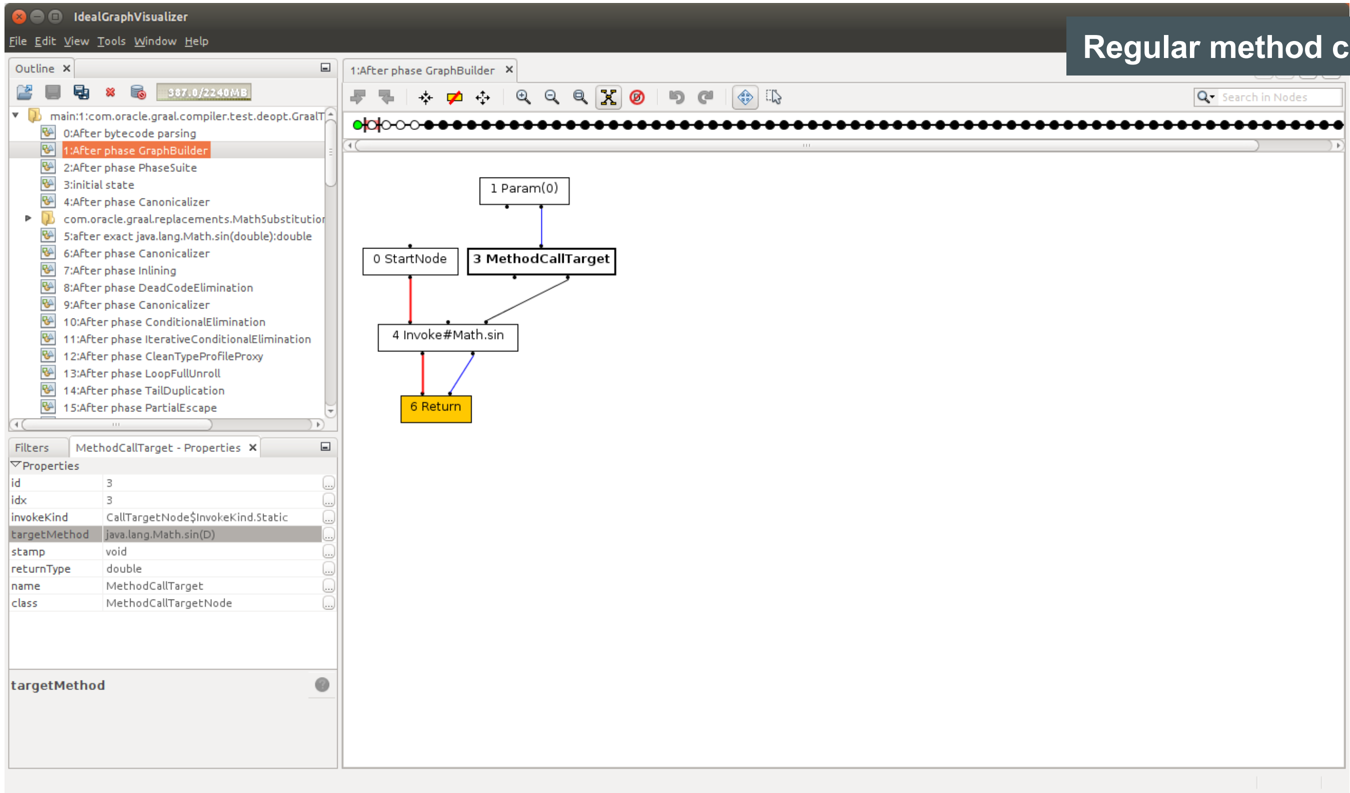
Command line to run example:

```
./mx.sh igv &
./mx.sh c1visualizer &
./mx.sh unittest -G:Dump= -G:MethodFilter=GraalTutorial.intrinsicUsage GraalTutorial#testIntrinsicUsage
```

C1Visualizer shows the LIR and generated machine code

Load the generated .cfg file with C1Visualzier

After Parsing



Regular method call to Math.sin()

Method Substitution

```
public class MathIntrinsicNode extends FloatingNode implements ArithmeticLIRLowerable {
  public enum Operation {LOG, LOG10, SIN, COS, TAN }

  @Input protected ValueNode value;
  protected final Operation operation;

  public MathIntrinsicNode(ValueNode value, Operation op) { ... }
  @NodeIntrinsic
  public static native double compute(double value, @ConstantNodeParameter Operation op);

  public void generate(NodeMappableLIRBuilder builder, ArithmeticLIRGenerator gen) { ... }
}
```

Node with node intrinsic shared several Math methods

LIR Generation

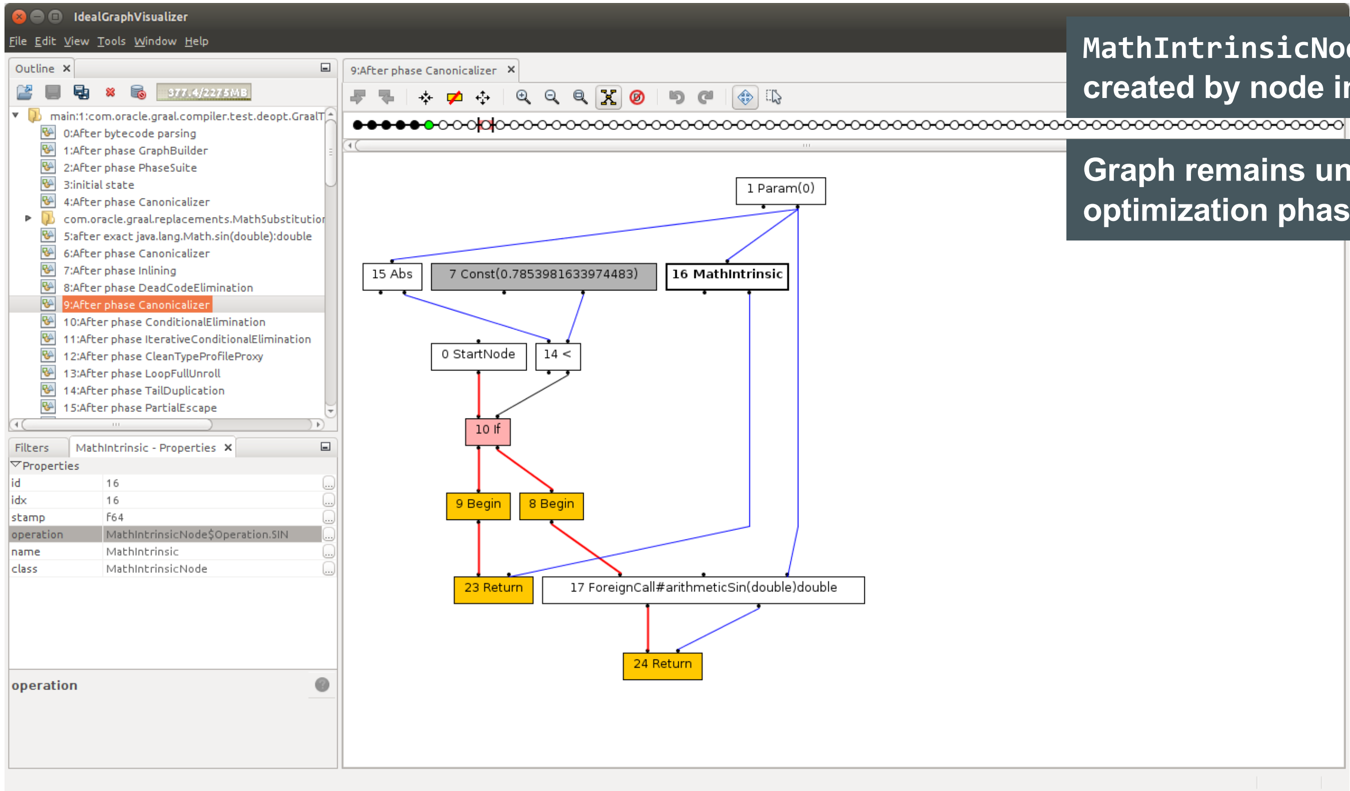
```
@ClassSubstitution(value = java.lang.Math.class)
public class MathSubstitutionsX86 {

  @MethodSubstitution(guard = UnsafeSubstitutions.GetAndSetGuard.class)
  public static double sin(double x) {
    if (abs(x) < PI_4) {
      return MathIntrinsicNode.compute(x, Operation.SIN);
    } else {
      return callDouble(ARITHMETIC_SIN, x);
    }
  }

  public static final ForeignCallDescriptor ARITHMETIC_SIN = new ForeignCallDescriptor("arithmeticSin", double.class, double.class);
}
```

Class that is substituted

The x86 instruction fsin can only be used for a small input values

Runtime call into the VM used for all other values

After Inlining the Substituted Method



`MathIntrinsicNode`, `AbsNode`, and `ForeignCallNode` are all created by node intrinsics

Graph remains unchanged throughout all further optimization phases

ORACLE

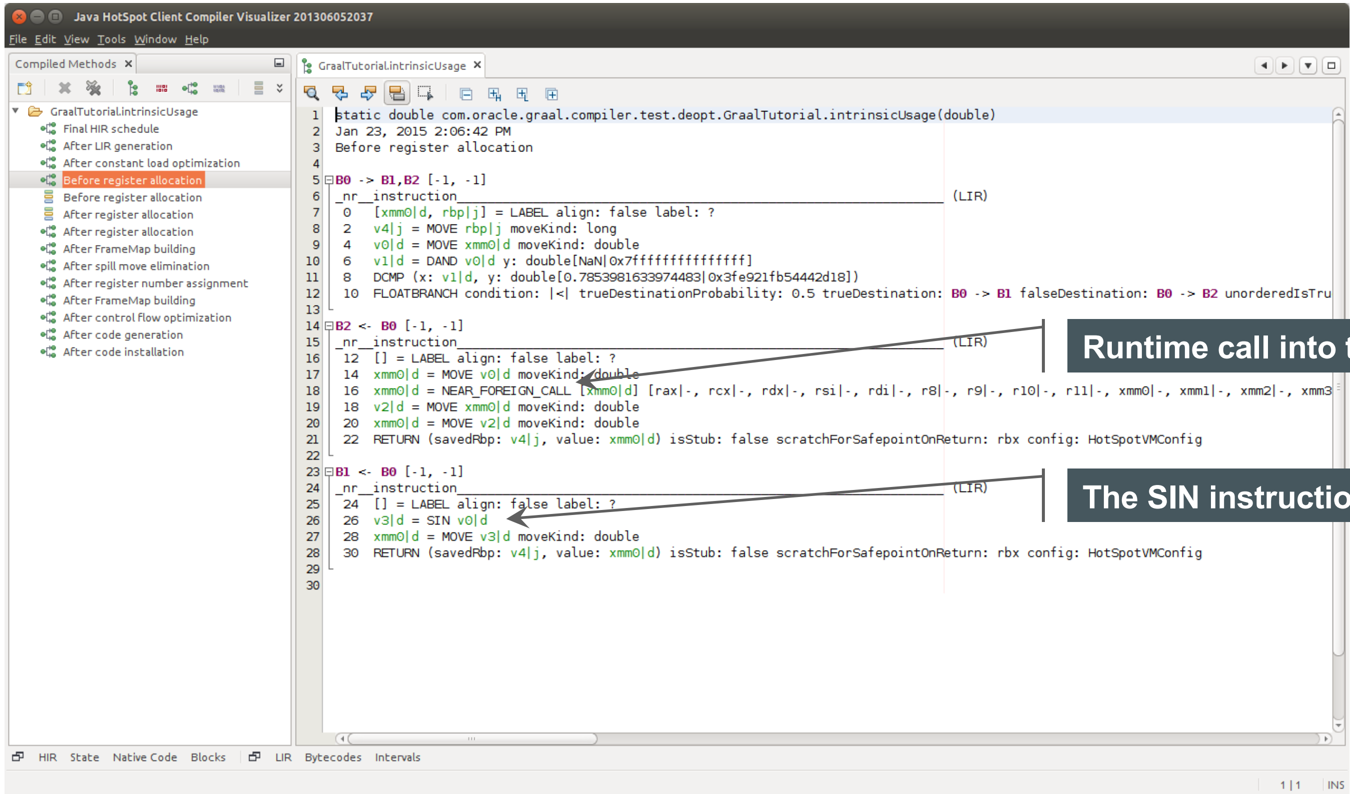
LIR Instruction

```
public class AMD64MathIntrinsicOp extends AMD64LIRInstruction {
  public enum IntrinsicOpcode  { SIN, COS, TAN, LOG, LOG10 }

  @Opcode private final IntrinsicOpcode opcode;
  @Def protected Value result;
  @Use protected Value input;

  public AMD64MathIntrinsicOp(IntrinsicOpcode opcode, Value result, Value input) {
    this.opcode = opcode;
    this.result = result;
    this.input = input;
  }

  @Override
  public void emitCode(CompilationResultBuilder crb, AMD64MacroAssembler masm) {
    switch (opcode) {
      case LOG:   masm.flog(asDoubleReg(result), asDoubleReg(input), false); break;
      case LOG10: masm.flog(asDoubleReg(result), asDoubleReg(input), true); break;
      case SIN:   masm.fsin(asDoubleReg(result), asDoubleReg(input)); break;
      case COS:   masm.fcos(asDoubleReg(result), asDoubleReg(input)); break;
      case TAN:   masm.ftan(asDoubleReg(result), asDoubleReg(input)); break;
      default:    throw GraalInternalError.shouldNotReachHere();
    }
  }
}
```

LIR uses annotation to specify input, output, or temporary registers for an instruction

Finally the call to the assembler to emit the bits

LIR Before Register Allocation



Runtime call into the VM (without JNI overhead)

The SIN instruction we are looking for

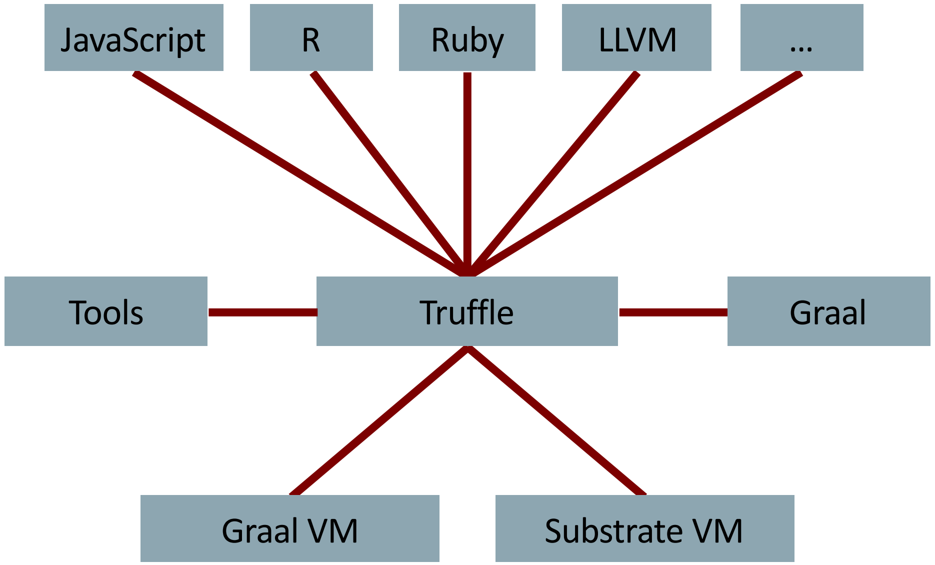
The ecosystem

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

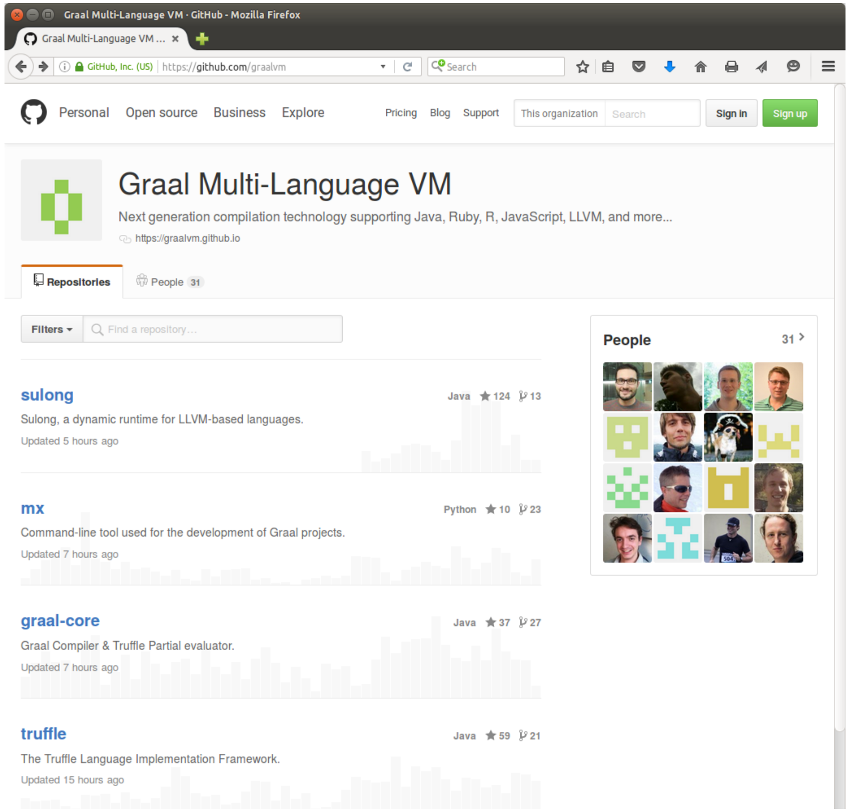
Low-footprint VM, also suitable for embedding

Truffle Language Projects

Some languages that we are aware of

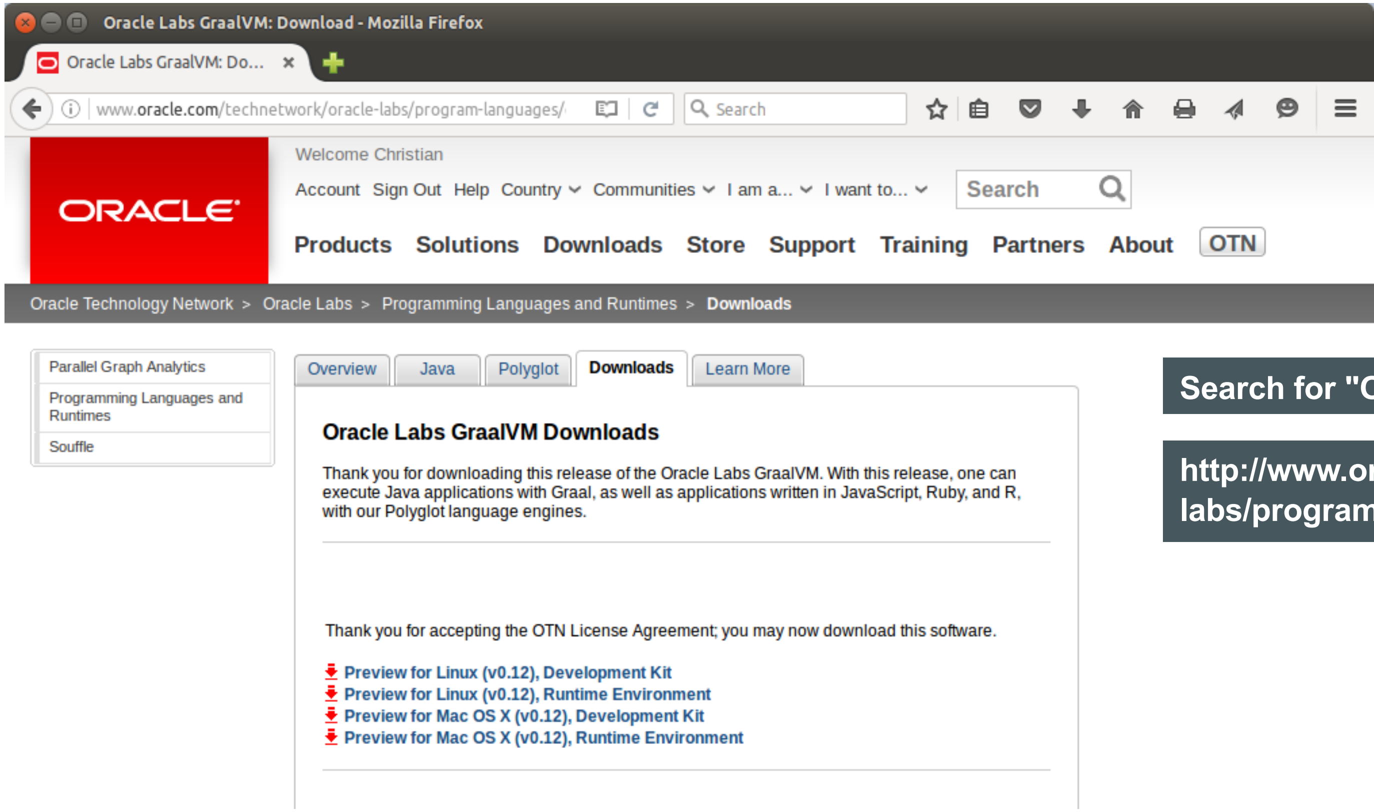
- JavaScript: JKU Linz, Oracle Labs
  – http://www.oracle.com/technetwork/oracle-labs/program-languages/
- Ruby: Oracle Labs, included in JRuby
  – Open source: https://github.com/jruby/jruby
- R: JKU Linz, Purdue University, Oracle Labs
  – Open source: https://github.com/graalvm/fastr
- Sulong (LLVM 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/

Open Source Code on GitHub



https://github.com/graalvm

Binary Snapshots on OTN



Search for "OTN Graal"

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

ORACLE

Results

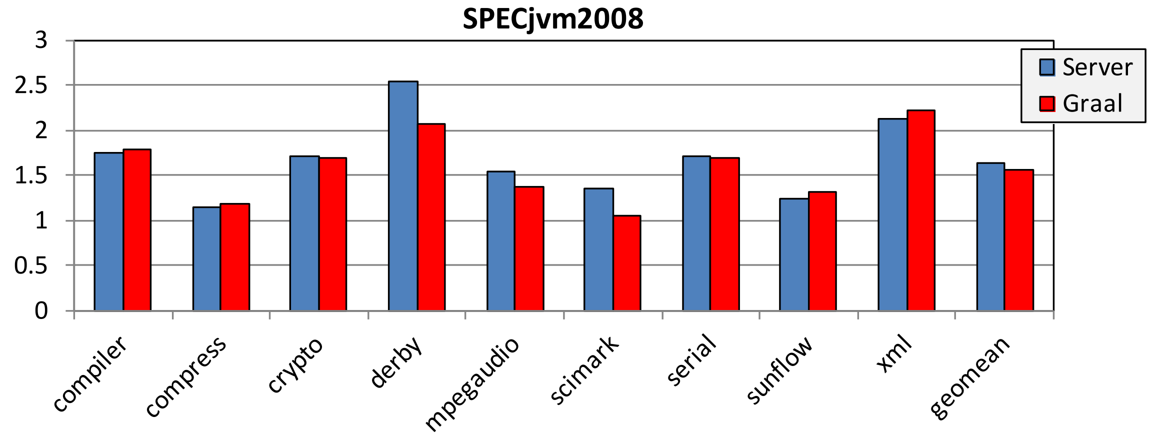
Performance Disclaimers

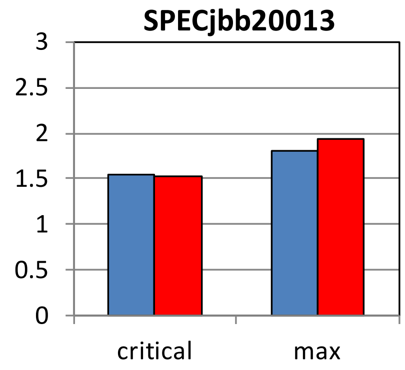
- All Truffle numbers reflect a development snapshot
  - Subject to change at any time (hopefully improve)
  - You have to know a benchmark to understand why it is slow or fast

- 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

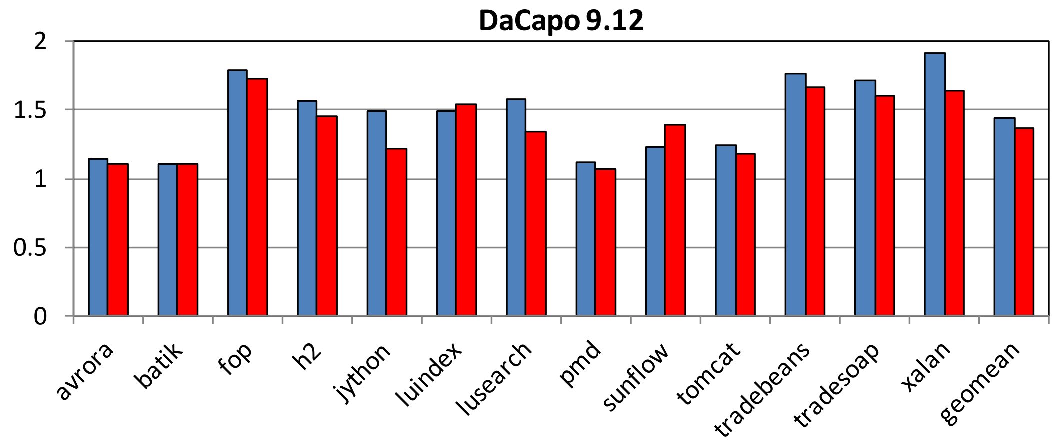
Graal Benchmark Results

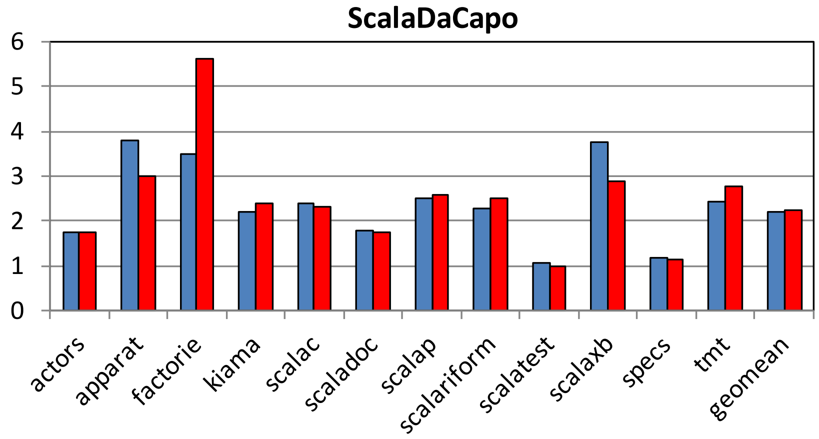




Higher is better, normalized to Client compiler.

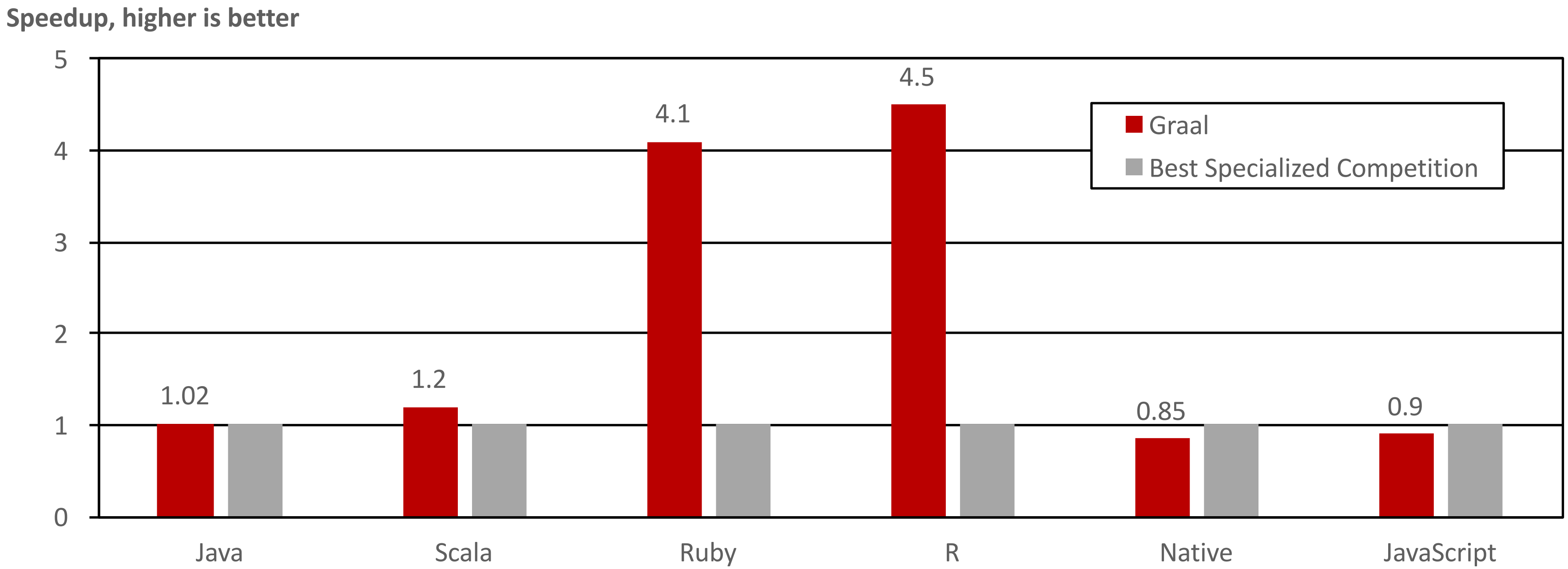
Results are not SPEC compliant, but follow the rules for research use.





ORACLE

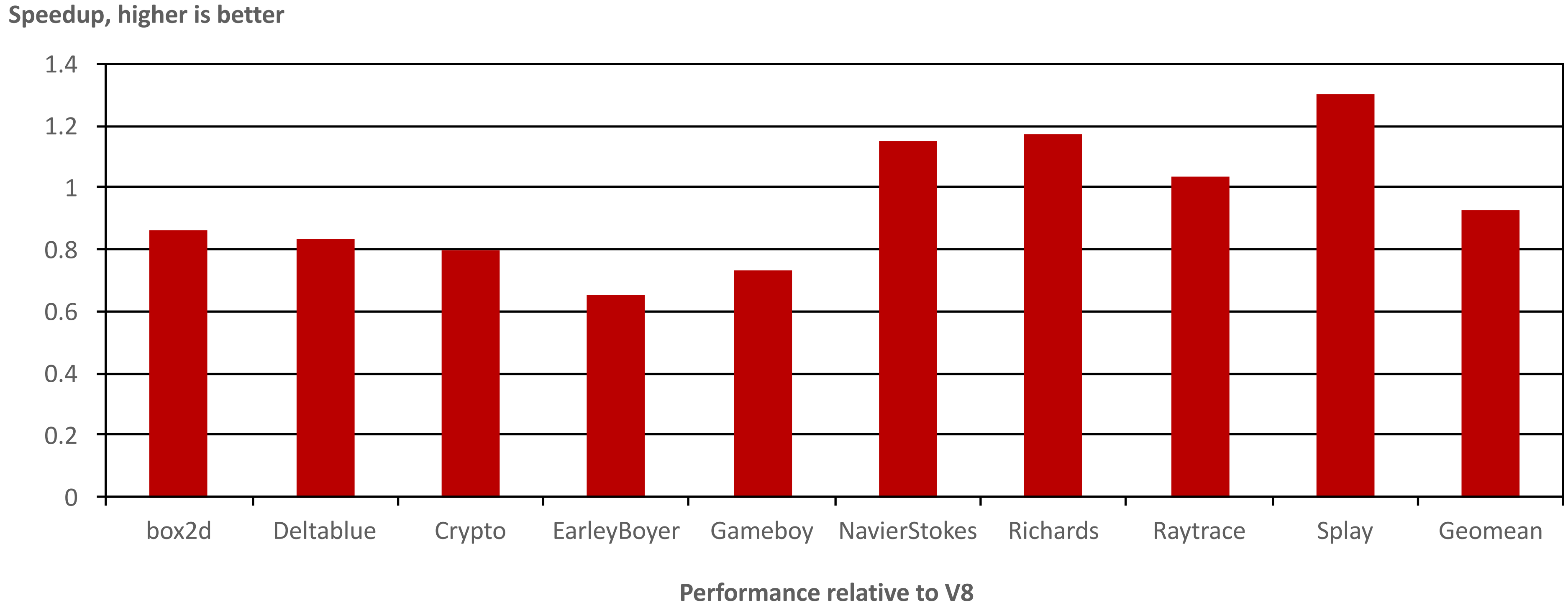
Performance: GraalVM Summary



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

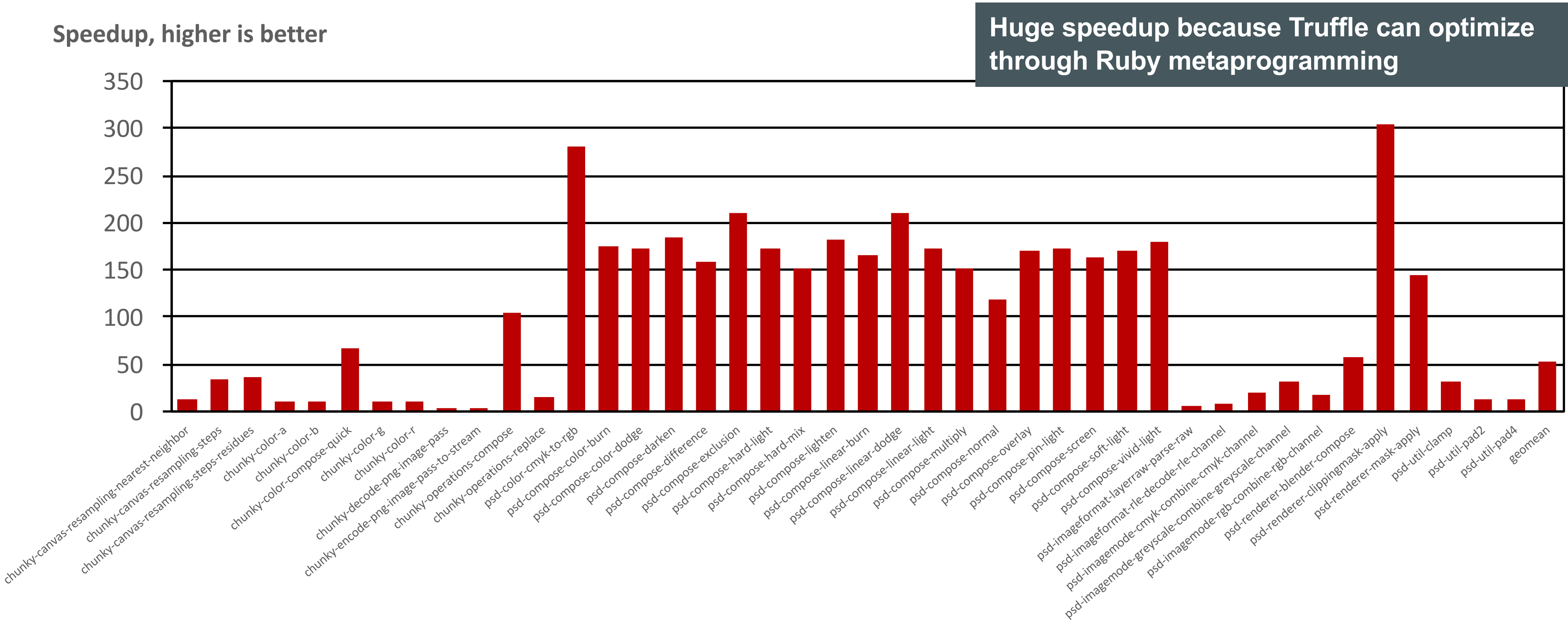
Performance: JavaScript

JavaScript performance: similar to V8



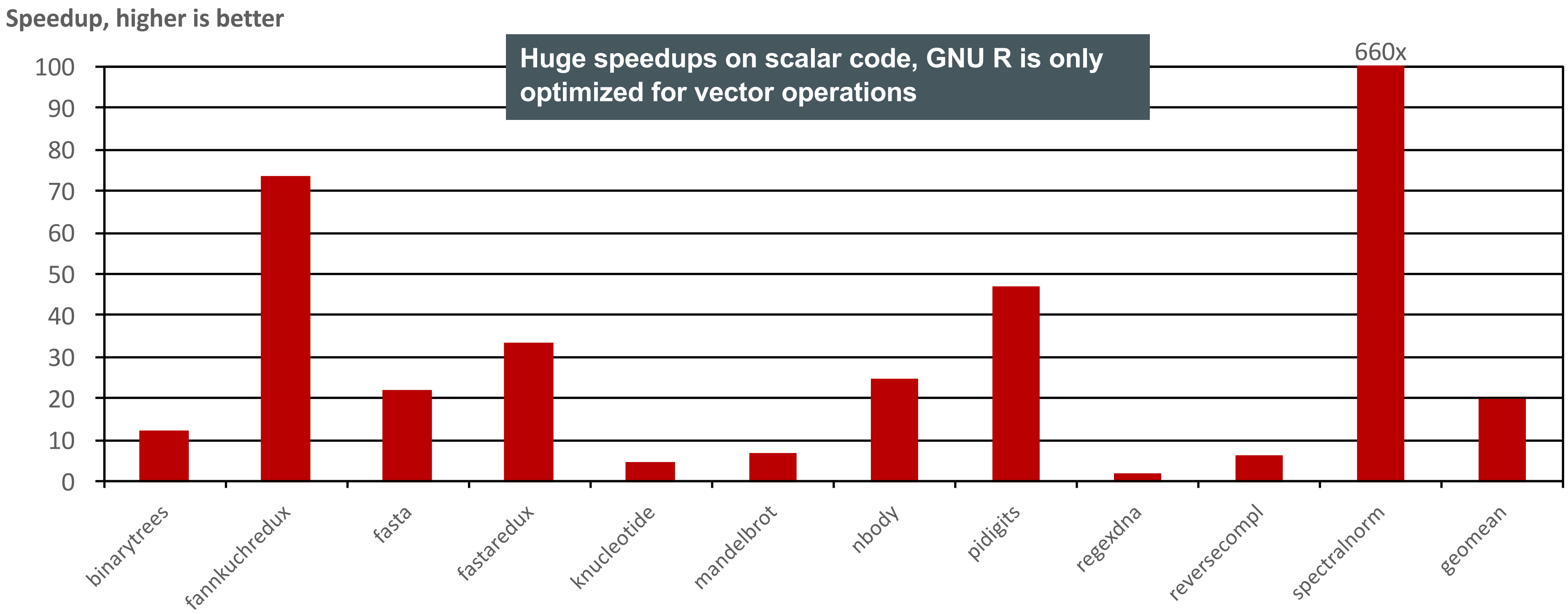
Performance: Ruby Compute-Intensive Kernels

Huge speedup because Truffle can optimize through Ruby metaprogramming



Performance relative to JRuby running with Java HotSpot server compiler

Performance: R with Scalar Code

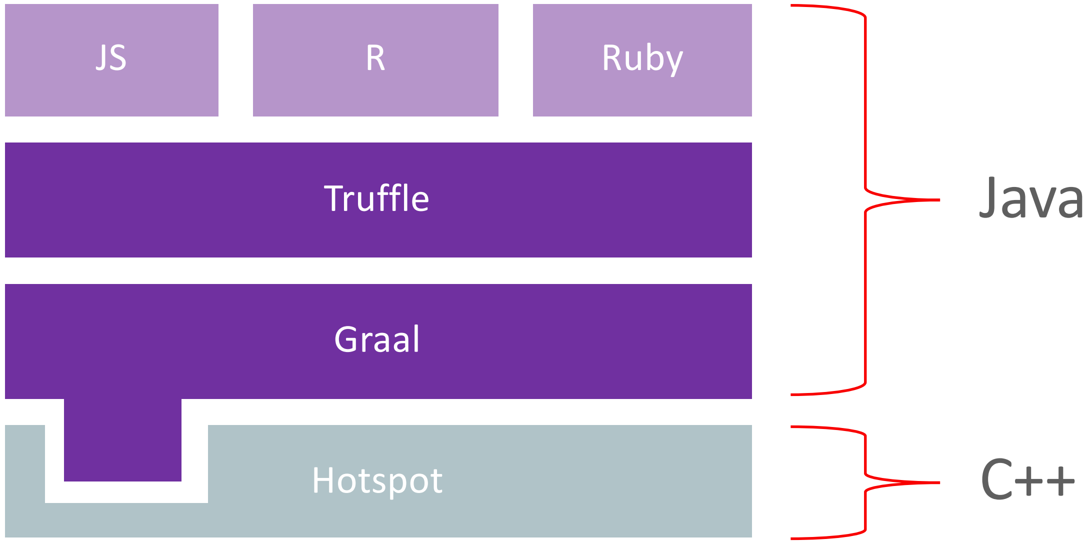


Performance relative to GNU R with bytecode interpreter

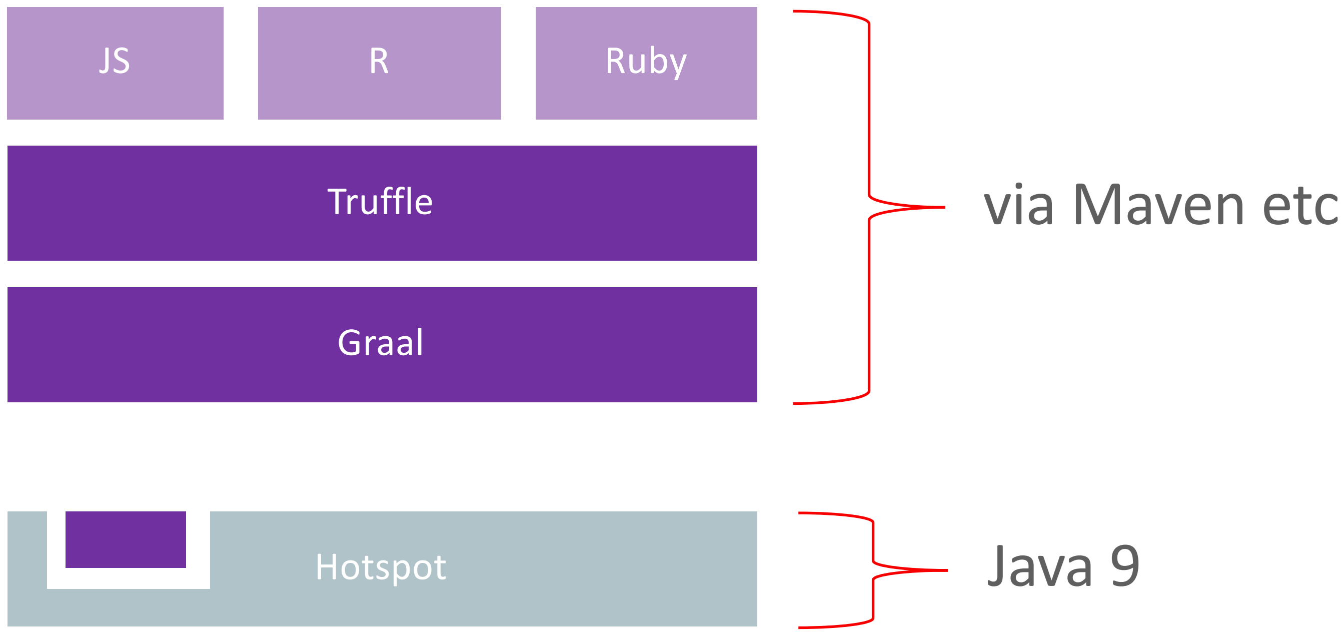
Will I be able to use Truffle and Graal for real?

ORACLE

JVMCI
(JVM Compiler Interface)



ORACLE


JS
R
Ruby
Truffle
Graal
via Maven etc
Hotspot
Java 9

Will I be able to use Truffle and Graal for real?

Acknowledgements

Oracle Labs
Danilo Ansaloni
Stefan Anzinger
Daniele Bonetta
Matthias Brantner
Laurent Daynès
Gilles Duboscq
Michael Haupt
Mick Jordan
Peter Kessler
Hyunjin Lee
David Leibs
Kevin Menard
Tom Rodriguez
Roland Schatz
Chris Seaton
Doug Simon
Lukas Stadler
Michael Van De Vanter

Oracle Labs (continued)
Adam Welc
Till Westmann
Christian Wimmer
Christian Wirth
Paul Wögerer
Mario Wolczko
Andreas Wöß
Thomas Würthinger

Oracle Labs Interns
Shams Imam
Stephen Kell
Gero Leinemann
Julian Lettner
Gregor Richards
Robert Seilbeck
Rifat Shariyar

Oracle Labs Alumni
Erik Eckstein
Christos Kotselidis

JKU Linz
Prof. Hanspeter Mössenböck
Benoit Daloze
Josef Eisl
Matthias Grimmer
Christian Häubl
Josef Haider
Christian Humer
Christian Huber
Manuel Rigger
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
Codrut Stancu
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

We’re interested in talking to people about

- Using Truffle or Graal directly
- Running Java programs on Graal
- Running JS, Ruby or R programs on our implementations
- Researching metaprogramming by modifying these implementations
- Internships for these projects and others

chris.seaton@oracle.com

ORACLE

Safe Harbor Statement

The preceding 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. Oracle reserves the right to alter its development plans and practices at any time, and 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.

Integrated Cloud

Applications & Platform Services

ORACLE®