One VM to Rule Them All

Christian Wimmer

VM Research Group, Oracle Labs
Safe Harbor Statement

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

JavaScript: One language to rule them all | VentureBeat
venturebeat.com/2011/...javascript-one_language-to-rule_them-all...
by Peter Yared - in 23 Google+ circles
Jul 29, 2011 - Why code in two different scripting languages, one on the client and one on the server? It's time for one language to rule them all. Peter Yared ...

[PDF] Python: One Script (Language) to rule them all - Ian Darwin
www.darwinsys.com/python/python4unix.pdf
Another Language? » Python was invented in 1991 by Guido van Rossum. » Named after the comedy troupe, not the snake. » Simple. » They all say that!

Q & Stuff: One Language to Rule Them All - Java
qstuff.blogspot.com/2005/10/one-language-to-rule-them-all-java.html
Oct 10, 2005 - One Language to Rule Them All - Java. For a long time I'd been hoping to add a scripting language to LibQ, to use in any of my (or other ...

Dart: one language to rule them all - MixIT 2013 - Slideshare
fr.slideshare.net/sdeleuze/dart-mixit2013en
DartSébastien Deleuze - @sdeleuze
One language to rule them all ...
One Language to Rule Them All?
Let’s ask Stack Overflow...

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

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

closed as not constructive by Tim, Bo Persson, Devon_C_Miller, Mark, Graviton Jan 17 at 5:58
“Write Your Own Language”

**Current situation**
- Prototype a new language
  - Parser and language work to build syntax tree (AST), AST Interpreter
- Write a “real” VM
  - In C/C++, still using AST interpreter, spend a lot of time implementing runtime system, GC, ...
- People start using it
- People complain about performance
  - Define a bytecode format and write bytecode interpreter
- Performance is still bad
  - Write a JIT compiler, improve the garbage collector

**How it should be**
- Prototype a new language in Java
  - Parser and language work to build syntax tree (AST)
  - Execute using AST interpreter
- People start using it
  - And it is already fast
Truffle System Structure

AST Interpreter for every language
Common API separates language implementation and optimization system

JavaScript  R  Ruby  Python  ...

Integrate with Java applications
Low-footprint VM, also suitable for embedding

Your language should be here!

Language agnostic dynamic compiler

Graal VM  Substrate VM

Truffle

Graal
Speculate and Optimize...

Node Rewriting for Profiling Feedback

Node Transitions

Uninitialized
Integer
String
Double
Generic

AST Interpreter Uninitialized Nodes

Compilation using Partial Evaluation

AST Interpreter Rewritten Nodes

Compiled Code
... and Deoptimize and Reoptimize!
More Details on Truffle Approach
https://wiki.openjdk.java.net/display/Graal/Publications+and+Presentations

One VM to Rule Them All

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

*Oracle Labs  †Institute for System Software, Johannes Kepler University Linz, Austria  ‡SS3 Lab, Purdue University
§{thomas.wuerthinger, christian.wimmer, doug.simon, mario.wolczko}@oracle.com
§§{woess, stadler, duboscq, christian.humer}@ssw.jku.at  ‡gr@purdue.edu

Abstract

Building high-performance virtual machines is a complex and expensive undertaking; many popular languages still have low-performance implementations. We describe a new approach to virtual machine (VM) construction that amortizes much of the effort in initial construction by allowing new languages to be implemented with modest additional effort. The approach relies on abstract syntax tree (AST) interpretation where a node can rewrite itself to a more specialized or more general node, together with an optimizing compiler. As Microsoft’s Common Language Runtime, the VM of the .NET framework [43]. These implementations can be characterized in the following way:

• Their performance on typical applications is within a small integer multiple (1-3x) of the best statically compiled code for most equivalent programs written in an unsafe language such as C.
• They are usually written in an unsafe, systems programming language (C or C++).
Truffle Language Projects

Some languages that we are aware of

- Ruby: Oracle Labs, experimental part of JRuby
  - Open source: https://github.com/jruby/jruby

- R: JKU Linz, Purdue University, Oracle Labs
  - Open source: https://bitbucket.org/allr/fastr

- Python: UC Irvine
  - Open source: https://bitbucket.org/ssllab/zippy/

- JavaScript: JKU Linz, Oracle Labs
  - http://www.oracle.com/technetwork/oracle-labs/program-languages/

- SOM (Smalltalk): Stefan Marr
  - Open source: https://github.com/smarr/TruffleSOM
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
  – Ruby: passing >70% of RubySpec language tests
    • About as complete as Topaz
    • JRuby passes about 86%
  – R: early prototype
  – Python: about 80% language completeness

• Benchmarks that are not shown
  – may not run at all, or
  – may not run fast
Performance: JavaScript
Performance: Ruby
psd.rb and chunky_png Ruby gems
Performance: Python

![Performance Graph]

- CPython 3.4
- Jython 2.7
- PyPy3 2.3.1
- ZipPy

[Bar chart details:]
- binarytrees
- frinkredux
- fasta
- mandelbrot
- meteor
- nbody
- piddigit
- spectrals
- float
- richards
- chaos
- deltablue
- go

[Graph axes:]
- X-axis: Benchmarks (binarytrees, frinkredux, etc.)
- Y-axis: Performance (time in seconds)
Acknowledgements

Oracle Labs
Danilo Ansaloni
Daniele Bonetta
Laurent Daynès
Erik Eckstein
Michael Haupt
Mick Jordan
Peter Kessler
Christos Kotselidis
David Leibs
Tom Rodriguez
Roland Schatz
Chris Seaton
Doug Simon
Lukas Stadler
Michael Van De Vanter
Adam Welc
Christian Wimmer
Christian Wirth
Paul Wögerer
Mario Wolczko
Andreas Wöß
Thomas Würthinger

Oracle Labs Interns
Stefan Anzinger
Shams Imam
Stephen Kell
Julian Lettnner
Gregor Richards
Rifat Shariyar

JKU Linz
Prof. Hanspeter Mössenböck
Gilles Dubosq
Matthias Grimmer
Christian Häubl
Josef Haider
Christian Humr
Christian Huber
Manuel Rigger
Bernhard Urban

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

LaBRI
Floréal Morandat
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

• Omitted language features
  – No floating point numbers
  – No object model: no run-time memory allocation
# Types

<table>
<thead>
<tr>
<th>SL Type</th>
<th>Values</th>
<th>Java Type in Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Arbitrary precision integer numbers</td>
<td>long for values that fit within 64 bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>java.lang.BigInteger on overflow</td>
</tr>
<tr>
<td>Boolean</td>
<td>true, false</td>
<td>boolean</td>
</tr>
<tr>
<td>String</td>
<td>Unicode characters</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>Function</td>
<td>Reference to a function</td>
<td>SLFunction</td>
</tr>
<tr>
<td>Null</td>
<td>null</td>
<td>SLNull.SINGLETON</td>
</tr>
</tbody>
</table>

Null is its own type because SL has no object 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

• 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
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
Hello World:

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

Simple loop:

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

First class functions:

```java
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);
}
```

Strings:

```java
function f(a, b) {
    return a + " < " + b + ": " + (a < b);
}
function main() {
    println(f(2, 4));
    println(f(2, "4"));
}
```

Function definition and redefinition:

```java
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();
}
```
SL Examples with Output

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

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

First class functions:
```sl
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);
}
```

Strings:
```sl
function f(a, b) {
    return a + " < " + b + ": " + (a < b);
}
function main() {
    println(f(2, 4));
    println(f(2, "4"));
}
```

Function definition and redefinition:
```sl
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();
}
```
Getting Started

Get and build the source code:

```bash
$ hg clone http://hg.openjdk.java.net/graal/graal
$ cd graal
$ ./mx.sh build
```

Run SL example program:

```bash
$ ./mx.sh sl graal/com.oracle.truffle.sl.test/tests/HelloWorld.sl
```

Generate Eclipse and NetBeans projects:

```bash
$ ./mx.sh ideinit
```

Import (at least) the following projects to work with SL:

![Package Explorer](image)

Use the "server" configuration when `mx` asks you

`mx` is our script to simplify building and execution
Simple Tree Nodes
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

```java
public abstract class Node implements Cloneable {
    protected Node(SourceSection sourceSection) { ... }

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

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

```java
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`
Blocks

```java
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:** The iteration of the children must be annotated with `@ExplodeLoop` and a final array.

**Rule:** A field for multiple child nodes must be annotated with `@Children` and a final array.
Return Statement: Inter-Node Control Flow

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

Rule: Exceptions used to model control flow extend ControlFlowException
Truffle DSL: Specialization and Node Rewriting
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")
    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;
    }
}
```

For all other specializations, guards are implicit based on method signature.
Code Generated by Truffle DSL (1)

Generated class hierarchy:

```
SLExpressionNode
  SLBinaryNode
    SLAddNode
      SLAddBaseNode
        SLAddBigIntegerNode
        SLAddLongNode
        SLAddPolymorphicNode
        SLAddStringToObjectNode
        SLAddUninitializedNode
```

Generated factory class:

```java
@GeneratedBy(SLAddNode.class)
public final class SLAddNodeFactory implements NodeFactory<SLAddNode> {

  public SLAddNode createNode(Object... arguments) { ... }

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

The parser creates a SLAddUninitializedNode, using the SLAddNodeFactory

SLAddPolymorphicNode is chains other node subclasses for polymorphic additions
Code Generated by Truffle DSL (2)

```java
@GeneratedBy(SLAddNode.class)
@NodeInfo(kind = Kind.SPECIALIZED, shortName = "+")
private final class SLAddLongNode extends SLAddBaseNode {
    public long executeLong(VirtualFrame frameValue) throws UnexpectedResultException {
        long leftNodeValue;
        try {
            leftNodeValue = leftNode.executeLong(frameValue);
        } catch (UnexpectedResultException ex) {
            Object rightNodeValue = rightNode.executeGeneric(frameValue);
            return SLTYPES.expectLong(executeAndSpecialize0(1, frameValue, ex.getResult(), rightNodeValue));
        }
        long rightNodeValue;
        try {
            rightNodeValue = this.rightNode.executeLong(frameValue);
        } catch (UnexpectedResultException ex) {
            return SLTYPES.expectLong(executeAndSpecialize0(1, frameValue, leftNodeValue, ex.getResult()));
        }
        try {
            return super.add(leftNodeValue, rightNodeValue);
        } catch (ArithmeticException ex) {
            return SLTYPES.expectLong(executeAndSpecialize0(1, frameValue, leftNodeValue, rightNodeValue));
        }
    }

    public Object executeGeneric(VirtualFrame frameValue) {
        try {
            return executeLong(frameValue);
        } catch (UnexpectedResultException ex) {
            return ex.getResult();
        }
    }
```
Type System Definition in Truffle DSL

```java
@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);
    }
}

@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 BigInteger executeBigInteger(VirtualFrame frame) ...
    public boolean executeBoolean(VirtualFrame frame) ...
    public String executeString(VirtualFrame frame ...) ...
    public SLFunction executeFunction(VirtualFrame frame) ...
    public SLNull executeNull(VirtualFrame frame) ...
}
```

Rule: One `execute()` method per type, in addition to the abstract `executeGeneric()` method.

Order of types is important: defines the order in which specializations are matched.

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

SLTypesGen is a generated subclass of SLTypes.
UnexpectedResultException

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

• But what to do when speculation was too optimistic?
  – Need to return a value with a type more general than the return type
  – Solution: return the value “boxed” in an UnexpectedResultException

• Exception handler performs node rewriting
  – Exception is thrown only once, so no performance bottleneck
Truffle DSL Workflow

1. Java Annotations (DSL Definition)
   - uses

2. Java Code with Node Specifications
   - compiles

3. Java Annotation Processor (DSL Implementation)
   - calls

4. iterates annotations

5. generates

6. compiles

7. generates

Executable

Generated Java Code for Specialized Nodes
Frames and Local Variables
Frame Layout

• In the interpreter, a frame is an object on the heap
  – Allocated in the function prologue
  – Passed around as parameter to execute() methods

• The compiler eliminates the allocation
  – No object allocation and object access
  – Guest language local variables have the same performance as Java local variables

• FrameDescriptor: describes the layout of a frame
  – A mapping from identifiers (usually variable names) to typed slots
  – Every slot has a unique index into the frame object
  – Created and filled during parsing

• Frame
  – Created for every invoked guest language function
Frame Management

• Truffle API only exposes frame interfaces
  – Implementation class depends on the optimizing system

• VirtualFrame
  – What you usually use: automatically optimized by the compiler
  – Must never be assigned to a field, or escape out of an interpreted function

• MaterializedFrame
  – A frame that can be stored without restrictions
  – Example: frame of a closure that needs to be passed to other function

• Allocation of frames
  – Factory methods in the class TruffleRuntime
Frame Management

```java
public interface Frame {
    FrameDescriptor getFrameDescriptor();
    Object[] getArguments();
    Object getValue(FrameSlot slot);
    boolean isType(FrameSlot slot);
    Type getType(FrameSlot slot) throws FrameSlotTypeException;
    void setType(FrameSlot slot, Type value);
    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

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

    @Specialization(rewriteOn = FrameSlotTypeException.class)
    protected final long readLong(VirtualFrame frame) throws FrameSlotTypeException {
        return frame.getLong(getSlot());
    }

    ...

    @Specialization(rewriteOn = FrameSlotTypeException.class)
    protected final Object readObject(VirtualFrame frame) throws FrameSlotTypeException {
        return frame.getObject(getSlot());
    }

    @Specialization(contains = {"readLong", "readBoolean", "readObject"})
    protected final Object read(VirtualFrame frame) {
        return frame.getValue(getSlot());
    }
}
```

Frame.getValue() never fails, also returns boxed primitive values.
Local Variables

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

  @Specialization(guards = "isLongKind")
  protected final long write(VirtualFrame frame, long value) {
    frame.setLong(getSlot(), value);
    return value;
  }

  ...

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

  protected final boolean isLongKind() {
    return isKind(FrameSlotKind.Long);
  }

  private boolean isKind(FrameSlotKind kind) {
    if (getSlot().getKind() == kind) {
      return true;
    } else if (getSlot().getKind() == FrameSlotKind.Illegal) {
      getSlot().setKind(kind);
      return true;
    } else {
      return false;
    }
  }
```
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
Compilation

**SL source code:**

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

**Run this example:**

```
./mx.sh sl -G:-TruffleBackgroundCompilation graal/com.oracle.truffle.sl.test/tests/LoopPrint.sl
```

-G:-TruffleBackgroundCompilation forces compilation in the main thread

```
./mx.sh igv &
./mx.sh sl -G:Dump= -G:-TruffleBackgroundCompilation graal/com.oracle.truffle.sl.test/tests/LoopPrint.sl
```

Add the flag -G:Dump= to dump compiled functions to IGV

**Machine code for loop:**

```
...  movq    rcx, 0x0
    jmp    L2:
L1:  safepoint
    mov     rsi, rcx
    addq    rsi, 0x1
    jo      L3:
    mov     rcx, rsi
L2:   cmp     rax, rcx
    jnle    L1:
    ...
L3:   call    deoptimize
```
Visualization Tools: IGV
Visualization Tools: IGV
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 if it 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 rewriting
  - One node per cached value
  - Build chain of multiple cache nodes
Polymorphic Inline Cache

Example of cache with length 2

After Parsing  ➔  1 Function  ➔  2 Functions  ➔  >2 Functions

- `SLInvokeNode`
- `SLUninitializedDispatch`
- `SLDirectDispatch`
- `SLGenericDispatch`
Invoke Node

```java
public final class SLInvokeNode extends SLExpressionNode {

    @Child protected SLExpressionNode functionNode;
    @Children protected final SLExpressionNode[] argumentNodes;
    @Child protected SLAbstractDispatchNode dispatchNode;

    @ExplodeLoop
    public Object executeGeneric(VirtualFrame frame) {
        SLFunction function = functionNode.executeFunction(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
final class SLUninitializedDispatchNode extends SLAbstractDispatchNode {

    protected Object executeDispatch(VirtualFrame frame, SLFunction function, Object[] arguments) {
        int depth = ...
        SLInvokeNode invokeNode = ...

        SLAbstractDispatchNode replacement;
        if (depth < INLINE_CACHE_SIZE) {
            SLAbstractDispatchNode next = new SLUninitializedDispatchNode();
            replacement = new SLDirectDispatchNode(next, function);
            replace(replacement);
        } else {
            replacement = new SLGenericDispatchNode();
            invokeNode.dispatchNode.replace(replacement);
        }

        return replacement.executeDispatch(frame, function, arguments);
    }
}

Separation of concerns: this node builds the inline cache chain
Inline Cache Node

```java
final class SLDirectDispatchNode extends SLAbstractDispatchNode {

    private final SLFunction cachedFunction;
    @Child private DirectCallNode callCachedTargetNode;
    @Child private SLAbstractDispatchNode nextNode;

    protected SLDirectDispatchNode(SLAbstractDispatchNode next, SLFunction cachedFunction) {
        this.cachedFunction = cachedFunction;
        this.callCachedTargetNode = Truffle.getRuntime().createDirectCallNode(cachedFunction.getCallTarget());
        this.nextNode = next;
    }

    protected Object executeDispatch(VirtualFrame frame, SLFunction function, Object[] arguments) {
        if (this.cachedFunction == function) {
            return callCachedTargetNode.call(frame, arguments);
        } else {
            return nextNode.executeDispatch(frame, function, arguments);
        }
    }
}
```

Rule: the cachedFunction must be a final field

Separation of concerns: this node performs the inline cache check and optimized dispatch
Generic Dispatch Node

```java
final class SLGenericDispatchNode extends SLAbstractDispatchNode {

    @Child private IndirectCallNode callNode = Truffle.getRuntime().createIndirectCallNode();

    protected Object executeDispatch(VirtualFrame frame, SLFunction function, Object[] arguments) {
        return callNode.call(frame, function.getCallTarget(), arguments);
    }
}
```

Separation of concerns: this is the always succeeding, but slow, fallback node
Example: Function Calls

Truffle framework code triggers compilation, function inlining, …
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:

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

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

Specialized AST for function `add()`:

```sl
SLRootNode
    bodyNode = SLFunctionBodyNode
        bodyNode = SLBlockNode
            bodyNodes[0] = SLWriteLocalVariableLongNode(name = "a")
            valueNode = SLReadArgumentNode(index = 0)
            bodyNodes[1] = SLWriteLocalVariableLongNode(name = "b")
            valueNode = SLReadArgumentNode(index = 1)
            bodyNodes[2] = SLReturnNode
                valueNode = SLAddLongNode
                    leftNode = SLReadLocalVariableLongNode(name = "a")
                    rightNode = SLReadLocalVariableLongNode(name = "b")
```
Function Inlining

• 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
  – The whole node tree of a function is duplicated
  – Call node is replaced with the root of the duplicated tree

• Benefits
  – Inlined tree is specialized separately
    • Result: context sensitive profiling information
  – All later optimizations see the big combined tree, without further work
    • Partial evaluation operates on the inlined tree

• Language-independent inlining logic
  – SLRootNode overrides methods that provide the AST copy that is inlined
  – SLRootNode keeps copy of the uninitialized (= non-specialized) AST
Function Inlining (1)

Frequently executed call
Function Inlining (2)
Function Inlining Nodes

```java
final class SLDirectDispatchNode extends SLAbstractDispatchNode {
    protected SLDirectDispatchNode(... SLFunction cachedFunction) {
        this.callCachedTargetNode = Truffle.getRuntime().createDirectCallNode(cachedFunction.getCallTarget());
        ...
    }
}
```

```java
public final class SLRootNode extends RootNode {
    @Child private SLExpressionNode bodyNode;
    private final SLExpressionNode uninitializedBodyNode;

    public SLRootNode(... SLExpressionNode bodyNode) {
        this.uninitializedBodyNode = NodeUtil.cloneNode(bodyNode);
        this.bodyNode = bodyNode;
    }

    public RootNode split() {
        return new SLRootNode(... NodeUtil.cloneNode(uninitializedBodyNode));
    }
}
```
Function Inlining Example

SL source code:

```javascript
function add(a, b) {
    return a + b;
}
function foo() {
    add(1, 2);
    add("x", "y") {
    }
}
```

AST before function inlining:
function add() called by both calls

```
SLReturnNode
value = SLAddPolymorphicNode
  left = SLReadLocalVariableObjectNode
  right = SLReadLocalVariableObjectNode
```

AST after function inlining:
function add() inlined for first call

```
SLReturnNode
value = SLAddLongNode
  left = SLReadLocalVariableLongNode
  right = SLReadLocalVariableLongNode
```

function add() inlined for second call

```
SLReturnNode
value = SLAddStringNode
  left = SLReadLocalVariableObjectNode
  right = SLReadLocalVariableObjectNode
```
Compilation with Inlined Function

SL source code without call:

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

Machine code for loop without call:

```
...             rcx, 0x0
movq           rcx, 0x0
jmp            L2:
L1: safepoint
  mov            rsi, rcx
  addq           rsi, 0x1
  jo             L3:
  mov            rcx, rsi
L2: cmp          rax, rcx
  jnle           L1:
...           call deoptimize
```

SL source code with call:

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

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

Machine code for loop with inlined call:

```
...             rcx, 0x0
movq           rcx, 0x0
jmp            L2:
L1: safepoint
  mov            rsi, rcx
  addq           rsi, 0x1
  jo             L3:
  mov            rcx, rsi
L2: cmp          rax, rcx
  jnle           L1:
...           call deoptimize
```

Compilation API
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
    • Hint that this method is not important for performance, i.e., hint to not inline it
  – 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
 GUARDS AND INTERPRETER PROFILING (1)

```java
public final class BranchProfile {
    @CompilationFinal private boolean visited;

    public void enter() {
        if (!visited) {
            CompilerDirectives.transferToInterpreterAndInvalidate();
            visited = true;
        }
    }
}

public final class SLIfNode extends SLStatementNode {
    private final BranchProfile thenTaken = BranchProfile.create();
    private final BranchProfile elseTaken = BranchProfile.create();

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

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

*transferToInterpreter*() does nothing when running in interpreter.
Guards and Interpreter Profiling (2)

```java
public final class CountingConditionProfile {
    @CompilationFinal private int trueCount;
    @CompilationFinal private int falseCount;

    public boolean profile(boolean value) {
        if (value) {
            if (trueCount == 0) {
                CompilerDirectives.transferToInterpreterAndInvalidate();
            }
            if (CompilerDirectives.inInterpreter()) {
                trueCount++;
            }
        } else {
            if (falseCount == 0) {
                CompilerDirectives.transferToInterpreterAndInvalidate();
            }
            if (CompilerDirectives.inInterpreter()) {
                falseCount++;
            }
        }
        return CompilerDirectives.injectBranchProbability((double) trueCount / (double) (trueCount + falseCount), value);
    }
}
```

```java
public final class SLIfNode extends SLStatementNode {

    private final ConditionProfile condition = ConditionProfile.createCountingProfile();

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

Create an assumption:

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

Check an assumption:

```java
void foo() {
    assumption.check();
    // Some code that is only valid if assumption is true.
}
```

Respond to an invalidated assumption:

```java
void bar() {
    try {
        foo();
    } catch (InvalidAssumptionException ex) {
        // Perform node rewriting, or other slow-path code to respond to change.
    }
}
```

Invalidate an assumption:

```java
assumption.invalidate();
```
Function Redefinition (2)

```java
public abstract class SLDefineFunctionBuiltin extends SLBuiltinNode {

    @Specialization
    public final String defineFunction(String code) {
        doDefineFunction(getContext(), code);
        return code;
    }

    @TruffleBoundary
    private static void doDefineFunction(SLContext context, String code) {
        Source source = Source.fromText(code, "[defineFunction]");
        Parser.parseSL(context, source);
    }
}
```

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

SL semantics: Functions can be defined and redefined at any time
Function Redefinition (3)

```java
public final class SLFunction {

    private RootCallTarget callTarget;
    private Assumption callTargetStable;

    protected void setCallTarget(RootCallTarget callTarget) {
        this.callTarget = callTarget;

        if (callTargetStable != null) {
            callTargetStable.invalidate();
        }
        callTargetStable = Truffle.getRuntime().createAssumption(name);
    }

    public RootCallTarget getCallTarget() {
        return callTarget;
    }

    public Assumption getCallTargetStable() {
        return callTargetStable;
    }
}
```

The utility class CyclicAssumption simplifies this code.
final class SLDirectDispatchNode extends SLAbstractDispatchNode {

    private final SLFunction cachedFunction;
    @Child private DirectCallNode callCachedTargetNode;
    private final Assumption cachedTargetStable;

    protected SLDirectDispatchNode(... SLFunction cachedFunction) {
        this.cachedFunction = cachedFunction;
        this.callCachedTargetNode = Truffle.getRuntime().createDirectCallNode(cachedFunction.getCallTarget());
        this.cachedTargetStable = cachedFunction.getCallTargetStable();
    }

    protected Object executeDispatch(VirtualFrame frame, SLFunction function, Object[] arguments) {
        if (this.cachedFunction == function) {
            try {
                cachedTargetStable.check();
                return callCachedTargetNode.call(frame, arguments);
            } catch (InvalidAssumptionException ex) {
                replace(nextNode);
            }
        }
        return nextNode.executeDispatch(frame, function, arguments);
    }
}
Compiler Assertions

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

• CompilerAsserts.compilationConstant()
  – Checks that the passed in value is a compile-time constant
  – 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

• Assertions are checked after aggressive compiler optimizations
  – Method inlining, constant folding, dead code elimination, escape analysis, ...
Trace the Compilation (1)

```
./mx.sh sl -G:-TruffleBackgroundCompilation -G:+TraceTruffleExpansion -G:+TraceTruffleCompilation
-G:+TraceTruffleCompilationDetails graal/com.oracle.truffle.sl.test/tests/LoopCall.sl
```

== running on Graal Truffle Runtime
[truffle] opt start        root add
OptimizedCallTarget.callRoot
  StableOptionValue.getValue
  OptimizedCallTarget.castArguments
  OptimizedCallTarget.getRootNode
  RootNode.getFrameDescriptor
  OptimizedCallTarget.callProxy
    SLRootNode.execute
      SLFunctionBodyNode.executeGeneric
        SLBlockNode.executeVoid
          SLExpressionNode.executeVoid
            SLExpressionNode.executeLong
            SLExpressionNode.executeVoid
            SLExpressionNode.executeLong
            SLReturnNode.executeVoid
              SLAddLongNode.executeGeneric
                SLReadLocalVariableLongNode.executeLong
                SLReadLocalVariableLongNode.executeLong
```

Useful to start finding performance problems, if you do not really have a clue what is wrong. You can then look at the tree, method by method, see what code is expanded, and optimize it with that information.

Output simplified for readability on slide
Trace the Compilation (2)

```
./mx.sh sl -G:-TruffleBackgroundCompilation -G:+TraceTruffleExpansion -G:+TraceTruffleExpansionSource -G:+TraceTruffleCompilation -G:+TraceTruffleCompilationDetails graal/com.oracle.truffle.sl.test/tests/LoopCall.sl
```

Makes the output "clickable" in the Eclipse Console view
### Print Function Inlining

```
./mx.sh sl -G:-TruffleBackgroundCompilation -G:+TraceTruffleInlining -G:+TraceTruffleCompilation -G:+TraceTruffleCompilationDetails graal/com.oracle.truffle.sl.test/tests/Inlining.sl
```

== running on Graal Truffle Runtime

<table>
<thead>
<tr>
<th>[truffle] inline start</th>
<th>root c</th>
<th>ASTSize 23 (0/0)</th>
<th>C/T 1000/ 3</th>
<th>L/T 1000/ 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>[truffle] inline success</td>
<td>root b</td>
<td>ASTSize 14 (0/0)</td>
<td>nodeCount 14/9</td>
<td>frequency 1.00</td>
</tr>
<tr>
<td>[truffle] inline success</td>
<td>root a</td>
<td>ASTSize 5 (0/0)</td>
<td>nodeCount 5/5</td>
<td>frequency 1.00</td>
</tr>
<tr>
<td>[truffle] inline done</td>
<td>root c</td>
<td>ASTSize 23 (0/0)</td>
<td>C/T 1000/ 3</td>
<td>L/T 1000/ 1000</td>
</tr>
<tr>
<td>[truffle] opt queued</td>
<td>root c</td>
<td>ASTSize 23 (0/0)</td>
<td>C/T 1000/ 3</td>
<td>L/T 1000/ 1000</td>
</tr>
<tr>
<td>[truffle] opt start</td>
<td>root c</td>
<td>ASTSize 23 (0/0)</td>
<td>C/T 1000/ 3</td>
<td>L/T 1000/ 1000</td>
</tr>
<tr>
<td>[truffle] opt done</td>
<td>root c &lt;opt&gt;</td>
<td>ASTSize 23 (0/0)</td>
<td>Time 200(185+15)ms CallNodes I 2/D 0 GraalNodes</td>
<td></td>
</tr>
<tr>
<td>[truffle] inline start</td>
<td>root g</td>
<td>ASTSize 117 (0/0)</td>
<td>C/T 1000/ 3</td>
<td>L/T 1000/ 1000</td>
</tr>
<tr>
<td>[truffle] inline success</td>
<td>root d</td>
<td>ASTSize 32 (0/0)</td>
<td>nodeCount 32/9</td>
<td>frequency 1.00</td>
</tr>
<tr>
<td>[truffle] inline success</td>
<td>root c &lt;opt&gt;</td>
<td>ASTSize 23 (0/0)</td>
<td>nodeCount 23/9</td>
<td>frequency 1.00</td>
</tr>
<tr>
<td>[truffle] inline success</td>
<td>root e</td>
<td>ASTSize 32 (0/0)</td>
<td>nodeCount 32/9</td>
<td>frequency 1.00</td>
</tr>
<tr>
<td>[truffle] inline success</td>
<td>root c &lt;opt&gt;</td>
<td>ASTSize 23 (0/0)</td>
<td>nodeCount 23/9</td>
<td>frequency 1.00</td>
</tr>
<tr>
<td>[truffle] inline success</td>
<td>root f</td>
<td>ASTSize 32 (0/0)</td>
<td>nodeCount 32/9</td>
<td>frequency 1.00</td>
</tr>
<tr>
<td>[truffle] inline success</td>
<td>root c &lt;opt&gt;</td>
<td>ASTSize 23 (0/0)</td>
<td>nodeCount 23/9</td>
<td>frequency 1.00</td>
</tr>
<tr>
<td>[truffle] inline done</td>
<td>root g</td>
<td>ASTSize 117 (0/0)</td>
<td>C/T 1000/ 3</td>
<td>L/T 1000/ 1000</td>
</tr>
<tr>
<td>[truffle] opt queued</td>
<td>root g</td>
<td>ASTSize 117 (0/0)</td>
<td>C/T 1000/ 3</td>
<td>L/T 1000/ 1000</td>
</tr>
<tr>
<td>[truffle] opt start</td>
<td>root g</td>
<td>ASTSize 117 (0/0)</td>
<td>C/T 1000/ 3</td>
<td>L/T 1000/ 1000</td>
</tr>
<tr>
<td>[truffle] opt done</td>
<td>root g &lt;opt&gt;</td>
<td>ASTSize 117 (0/0)</td>
<td>Time 100(98+2)ms CallNodes I 12/D 0 GraalNodes</td>
<td></td>
</tr>
</tbody>
</table>
```

---

**Copyright © 2014, Oracle and/or its affiliates. All rights reserved. | 76**
### Print CallTarget Profile

```
./mx.sh sl -G:-TruffleBackgroundCompilation -G:+TruffleCallTargetProfiling
graal/com.oracle.truffle.sl.test/tests/Inlining.sl
```

<table>
<thead>
<tr>
<th>Call Target</th>
<th>Call Count</th>
<th>Calls Sites Inlined / Not Inlined</th>
<th>Node Count</th>
<th>Inv</th>
</tr>
</thead>
<tbody>
<tr>
<td>root g</td>
<td>2200</td>
<td>12 / 0</td>
<td>92</td>
<td>0</td>
</tr>
<tr>
<td>root f</td>
<td>1300</td>
<td>3 / 0</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>root e</td>
<td>1300</td>
<td>3 / 0</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>root c</td>
<td>1200</td>
<td>2 / 0</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>root b</td>
<td>1100</td>
<td>1 / 0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>root a</td>
<td>1000</td>
<td>0 / 0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>root d</td>
<td>999</td>
<td>0 / 1</td>
<td>8</td>
<td>0   int</td>
</tr>
<tr>
<td>root main</td>
<td>1</td>
<td>0 / 1</td>
<td>26</td>
<td>0   int</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>9100</strong></td>
<td><strong>21</strong> / 2</td>
<td><strong>209</strong></td>
<td>0</td>
</tr>
</tbody>
</table>


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
  – Try using @CompilationFinal if the Java final is too restrictive

• Use microbenchmarks to assess and track performance of specializations
  – Ensure and assert that you end up in the expected specialization
Truffle Mindset: Frames

• Use VirtualFrame, and ensure it does not escape
  – Sometimes, you get strangely looking error messages about escaping frames
  – 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 probably slower
Object Layout
Dynamic Object

```java
public class YourLanguageObject {  
    Shape shape;
    long primitiveStorageLocation1;
    ...
    long primitiveStorageLocationN;
    Object objectStorageLocation1;
    ...
    Object objectStorageLocationN;
    long[] primitiveStorageExtension; // Allocated only when necessary.
    Object[] objectStorageExtension; // Allocated only when necessary.
}
```

```java
public class Shape {  
    Shape parent;
    String propertyName;
    Type propertyType;
    int propertyIndex;
}
```

Most guest language objects require only one Java object.
Object Layout Transitions (1)

```javascript
var x = {};
x.foo = 0;
x.bar = 0;
// + subtree A
```
Object Layout Transitions (2)

```javascript
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)

```javascript
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
```
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
  – Deoptimization used for write access of frames down the stack
Stack Walking

public abstract class SLStackTraceBuiltin extends SLBuiltinNode {

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

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

    return str.toString();
}

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

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

TruffleRuntime provides stack walking

FrameInstance is a handle to a guest language frame
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 deoptimization

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

```java
public abstract class SLHelloEqualsWorldBuiltIn extends SLBuiltinNode {

    @Specialization
    @TruffleBoundary
    public String change() {
        FrameInstance frameInstance = Truffle.getRuntime().getCallerFrame();
        Frame frame = frameInstance.getFrame(FrameAccess.READ_WRITE, false);
        FrameSlot slot = frame.getFrameDescriptor().findOrAddFrameSlot("hello");
        frame.setObject(slot, "world");
        return "world";
    }
}
```

A constructed example, but the mechanism is generally usable
Substrate VM
Substrate VM

• Goal
  – Run Truffle languages without the overhead of a Java VM

• Approach
  – Ahead-of-time compile the Java bytecodes to machine code
  – Build standard Linux / MacOS executable
Substrate VM: Execution Model

**Static Analysis**
- Truffle Language
- JDK
- Substrate VM

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

**Ahead-of-Time Compilation**
- Reachable methods, fields, and classes

Application running without dependency on JDK and without Java class loading

**Output**
- Machine Code
- Initial Heap
- DWARF Info
- ELF / MachO Binary
Substrate VM: Startup Performance

Running Ruby “Hello World”

![Execution Time and Memory Footprint Diagram]

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

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

Substrate VM eliminates the Java VM startup overhead: Orders of magnitude for time and memory
Debugging Tools
### “Write Your Own Language and Debugger”

<table>
<thead>
<tr>
<th>Current situation</th>
<th>How it should be</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prototype a new language</strong></td>
<td><strong>Prototype a new language in Java</strong></td>
</tr>
<tr>
<td>Parser and language work to build syntax tree (AST), AST Interpreter</td>
<td>Parser and language work to build syntax tree (AST)</td>
</tr>
<tr>
<td><strong>Write a “real” VM</strong></td>
<td>Execute using AST interpreter</td>
</tr>
<tr>
<td>In C/C++, still using AST interpreter, spend a lot of time implementing runtime system, GC, ...</td>
<td>People start using it</td>
</tr>
<tr>
<td><strong>People start using it</strong></td>
<td>And it is already fast</td>
</tr>
<tr>
<td><strong>People complain about performance</strong></td>
<td><strong>And it has a debugger</strong></td>
</tr>
<tr>
<td>Define a bytecode format and write bytecode interpreter</td>
<td>People want supporting tools</td>
</tr>
<tr>
<td><strong>People want supporting tools</strong></td>
<td>Write a debugger (really?)</td>
</tr>
<tr>
<td>Write a debugger (really?)</td>
<td>Use <code>printf</code> (more likely)</td>
</tr>
</tbody>
</table>
System Structure **Extended for Debugging**

- AST Interpreter for every language
- Common API between language implementation and debugging framework and optimization system
- Integrate with Java applications
- Low-footprint VM, also suitable for embedding
- Language agnostic dynamic compiler
- Integrate with Java applications

- **JavaScript**
- **R**
- **Ruby**
- **Python**
- **...**

Your language should be here!
Proxy Nodes as Evaluation “Hooks”

Unmodified Truffle AST

Insert Proxies

Debuggable Truffle AST

Proxies intercept execution if needed, e.g., to “break”

Marks these nodes as “statements” for debugging

Proxied nodes indexed by source location

The language implementer:
• Decides which nodes to proxy
• Tags nodes with desired debugging behavior
• Adds other language-specific behavior as needed

Seaton, Van De Vanter, Haupt: Debugging at Full Speed; Accepted at DYLA’14
Proxy Nodes

• Debugging Proxies are
  – Legitimate Truffle AST nodes
  – Transparent to execution semantics of program
    • Default behavior is to pass through all invocations
  – Independent of other platform services (mostly)
  – Compiled to no-ops when not active
  – Compiled into fast-path when active
    • Useful for breakpoint conditions in long-running code
  – Reconstructed through deoptimization, simplifying user interaction
  – A generalization of the approach used for the Self Debugger†
    • Set a breakpoint by modifying program, then reoptimize

Summary
Your Language?

http://openjdk.java.net/projects/graal/
graal-dev@openjdk.java.net

$ hg clone http://hg.openjdk.java.net/graal/graal
$ cd graal
$ ./mx --vm server build
$ ./mx ideinit
$ ./mx --vm server unitest SumTest

More Installation Instructions:
https://wiki.openjdk.java.net/display/Graal/Instructions

Truffle API Resources:
https://wiki.openjdk.java.net/display/Graal/Truffle+FAQ+and+Guidelines

Truffle API License: GPLv2 with Classpath Exception
Hardware and Software
Engineered to Work Together