Truffle/Graal:
From Interpreters to Optimizing Compilers via Partial Evaluation

Jonathan Aldrich
Carnegie Mellon University

Many slides from Oracle’s PLDI 2016 and 2017 tutorials on Truffle/Graal—marked where they occur.
Interpreters for Prototyping

• Writing optimizing compilers is hard
  • Many complex optimization algorithms
  • Relies on knowledge of architecture and performance characteristics
  • See 15-411/15-611, 15-745

• So when we prototype with interpreters
  • Easy to write
  • Easy to change when the language changes

• Unfortunately, interpreters are slow
  • Especially if you make them simple!
  • Could be 2 orders of magnitude slower than an optimizing compiler
  • Is there any way to make an interpreter faster?
From Interpreter to Compiler

• Given
  • An interpreter for guest language G written in host language H
    • “if you see an add expression, add the two arguments”
  • A program in language G
    • “add(add(x, y), z)”

• What if we “run” the interpreter on the program?
  • Whenever we get to actual input data, we’ll stop evaluating there (but keep going elsewhere)
  • “add(add(x, y), z)” $\rightarrow$ $x+y+z$
  • We have essentially “compiled” the language

• This is called Partial Evaluation
  • When applied in the context of interpreters and programs, it’s called the First Futamura Projection (after Futamura, who proposed it in 1971)
Example: Partial Evaluation

class ExampleNode {
   @CompilationFinal boolean flag;

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

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

mov  rax, 42
ret

Object value of this
ExampleNode
flag: true

normal compilation
of method foo()

partial evaluation
of method foo()
with known parameter this

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

@CompilationFinal field is treated like a final field during partial evaluation
Introduction to Partial Evaluation

abstract class Node {
    abstract int execute(int[] args);
}

class AddNode extends Node {
    final Node left, right;

    AddNode(Node left, Node right) {
        this.left = right; this.right = right;
    }

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

class Arg extends Node {
    final int index;
    Arg(int i) {this.index = i;}

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

int interpret(Node node, int[] args) {
    return node.execute(args);
}

// Sample program (arg[0] + arg[1]) + arg[2]
sample = new Add(new Add(new Arg(0), new Arg(1)), new Arg(2));
int interpret(Node node, int[] args) {
    return node.execute(args);
}

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

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

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

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

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

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

int interpretSample(int[] args) {
    return args[0] + args[1] + args[2];
}
Challenge: how to get high-performance code? (1)

• With naïve Futamura projection / partial evaluation, code size explodes

• Real implementation of “+” in a dynamic language is something like:

```java
if (arg1 instanceof Int && arg2 instanceof Int)
    return arg1+arg2;
else if (arg1 instanceof String || arg2 instanceof String)
    return strcat(toString(arg1), toString(arg2))
else
    throw addError
```

• This is a lot of code to generate every time we see a + operator
Challenge: how to get high-performance code? (2)

• Alternative: don’t partially evaluate inside complex operations

\[
\text{add}(\text{add}(x, y), z)
\]

is transformed to

\[
\text{let } t = \text{doAdd}(x, y) \text{ in }
\text{doAdd}(z)
\]

• But now we lose many of the benefits of partial evaluation
  • We want add to turn into \(~2\) instructions, not an expensive function call.
  • Plus we can’t do further optimization easily – e.g. constant-fold when \(x\) and \(y\) are constants.
Challenge: how to get high-performance code? (3)

• Assume y is the constant 1, z is an Int, and x is probably an Int
• What we want is to translate “add(add(x, y), z)” into:

```java
// guess that x is an Int, but check to make sure
if (!x instanceof Int)) goto do_it_slowly
x+1+z
```

• We can figure out the y is the constant 1 with partial evaluation
• How do we know that z is definitely an Int?
• How can we guess that x is probably an Int?
Profile-Based Optimizing Interpreters

- Run each function in the Guest language a few times
- Gather profile information for each node in the AST
  - Call nodes: what functions are called?
  - Operation nodes: what are the datatypes?
- Specialize the interpreter
  - Replace unspecialized nodes with specialized ones
    - E.g. replace unspecializeAdd() with intAdd()
  - Each specialized node has a guard
    - intAdd(): check that my arguments are ints
    - If the guard fails, intAdd() knows how to replace itself with genericAdd()
- Partially evaluate the specialization
  - Generates optimized code
  - This is speculative – must include a check, and a hook to jump back to the interpreter to de-specialize
The Truffle Idea

Collect profiling feedback

Optimize using partial evaluation assuming stable profiling feedback

Deoptimize if profiling feedback is invalid and reprofile
Stability

![Graph showing stability over function invocations for different languages.](image-url)
Example: Transfer to Interpreter

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

transferToInterpreter() is a call into the VM runtime that does not return to its caller, because execution continues in the interpreter.
Example: Partial Evaluation and Transfer to Interpreter

class ExampleNode {
    @CompilationFinal boolean minValueSeen;
    int negate(int value) {
        if (value == Integer.MIN_VALUE) {
            if (!minValueSeen) {
                transferToInterpreterAndInvalidate();
                minValueSeen = true;
            }
            throw new ArithmeticException();
        }
        return -value;
    }
}

// parameter value in eax
cmp eax, 0x80000000
jz L1
neg eax
ret
L1: mov [rsp + 24], eax
call transferToInterpreterAndInvalidate

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

ExampleNode
minValueSeen: false

second partial evaluation

ExampleNode
minValueSeen: true

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

if compiled code is invoked with minimum int value:
1) transfer back to the interpreter
2) invalidate the compiled code

partial evaluation
of method negate() with known parameter this

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

ExampleNode
minValueSeen: true

partial evaluation of method negate() with known parameter this

ExampleNode
minValueSeen: false

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

Create an assumption:

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

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

Check an assumption:

```java
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:

```java
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
        }
    }
}

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

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

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
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

<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 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>Object</td>
<td>key-value store</td>
<td>DynamicObject</td>
</tr>
<tr>
<td>Null</td>
<td>null</td>
<td>SLNull.SINGLETON</td>
</tr>
</tbody>
</table>

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

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

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

• C-like syntax for control flow
  – if, while, break, continue, return

• Operators
  – +, -, *, /, ==, !=, <, <=, >, >=, &&, ||, [ ], ( )
  – + is defined on String, performs String concatenation
  – && and || have short-circuit semantics
  – . or [ ] for property access

• Literals
  – Number, String, Function

• Builtin functions
  – println, readln: Standard I/O
  – nanoTime: to allow time measurements
  – defineFunction: dynamic function redefinition
  – stacktrace, helloEqualsWorld: stack walking and stack frame manipulation
  – new: Allocate a new object without properties
Parsing

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

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

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

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

**Hello World:**

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

**Simple loop:**

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

**First class functions:**

```javascript
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:**

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

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

**Objects:**

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

**Function definition and redefinition:**

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

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

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

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

```java
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

```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`
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.
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

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

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

```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 SLReturnException extends ControlFlowException {
    private final Object result;
    ...
}
```

```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;
    }
}
```
Exceptions for Inter-Node Control Flow

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

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

Inter-Node Control Flow

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

SLFunctionBodyNode

SLBlockNode

SLReturnNode

bodyNode

... valueNode

...
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:

```java
@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.
The generated code can and will change at any time
Type System Definition in Truffle DSL

```java
@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) ...
}
```

```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);
    }
}
```

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

Not shown in slide: Use `@TypeCheck` and `@TypeCast` to customize type conversions.
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 compiler (javac, Eclipse, …) calls
4. Java Annotation Processor (DSL Implementation) iterates annotations
5. Generates Java Code for Specialized Nodes generates
6. Generated Java Code for Specialized Nodes compiles
7. Executable generates
Frames and Local Variables
Frame Layout

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

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

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

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

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

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

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

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

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

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

    Object getValue(FrameSlot slot);

    MaterializedFrame materialize();
}
```

Frames support all Java primitive types, and `Object`

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

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

```java
@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

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

    protected abstract FrameSlot getSlot();

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

    protected boolean isLong(VirtualFrame frame) {
        return getSlot().getKind() == FrameSlotKind.Long;
    }

    ...

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

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

The guard ensure the frame slot contains a primitive long value.

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

• Automatic partial evaluation of AST
  – Automatically triggered by function execution count

• Compilation assumes that the AST is stable
  – All @Child and @Children fields treated like final fields

• Later node rewriting invalidates the machine code
  – Transfer back to the interpreter: “Deoptimization”
  – Complex logic for node rewriting not part of compiled code
  – Essential for excellent peak performance

• Compiler optimizations eliminate the interpreter overhead
  – No more dispatch between nodes
  – No more allocation of VirtualFrame objects
  – No more exceptions for inter-node control flow
Truffle Compilation API

• Default behavior of compilation: Inline all reachable Java methods

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

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

    When compiling, the output stream is a constant
Compiler Assertions

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

• CompilerAsserts.partialEvaluationConstant()
  – Checks that the passed in value is a compile-time constant early during partial evaluation

• CompilerAsserts.compilationConstant()
  – Checks that the passed in value is a compile-time constant (not as strict as partialEvaluationConstant)
  – Compiler fails with a compilation error if the value is not a constant
  – When the assertion holds, no code is generated to produce the value

• CompilerAsserts.neverPartOfCompilation()
  – Checks that this code is never reached in a compiled method
  – Compiler fails with a compilation error if code is reachable
  – Useful at the beginning of helper methods that are big or rewrite nodes
  – All code dominated by the assertion is never compiled
Compilation

**SL source code:**

```javascript
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
Truffle Mindset: Frames

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

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

• Use a MaterializedFrame if a VirtualFrame is too restrictive
  – But keep in mind that access is slower
Function Calls
Polymorphic Inline Caches

• Function lookups are expensive
  – At least in a real language, in SL lookups are only a few field loads

• Checking whether a function is the correct one is cheap
  – Always a single comparison

• Inline Cache
  – Cache the result of the previous lookup and check that it is still correct

• Polymorphic Inline Cache
  – Cache multiple previous lookups, up to a certain limit

• Inline cache miss needs to perform the slow lookup

• Implementation using tree specialization
  – Build chain of multiple cached functions
Example: Simple Polymorphic Inline Cache

```java
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 `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

Compile time constants are usually the starting point for more constant folding
Polymorphic Inline Cache for Function Dispatch

Example of cache with length 2

- After Parsing
- 1 Function
- 2 Functions
- >2 Functions

The different dispatch nodes are for illustration only, the generated code uses different names.
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);
    }
}
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):

```java
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):

```java
assumption1.check();
if (function instanceof SLFunction) {
    if (function == cachedFunction) {
        callNode.call(frame, arguments);
    }
}
```

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

Partial evaluation can go across function boundary (function inlining) because callNode with its callTarget is final
Language Nodes vs. Truffle Framework Nodes

Truffle framework code triggers compilation, function inlining, …
Function Redefinition (1)

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

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

• Result
  – No overhead when calling a function
Function Redefinition (2)

```java
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
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:

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

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

Specialized AST for function `add()`:

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

• Function inlining is one of the most important optimizations
  – Replace a call with a copy of the callee

• Function inlining in Truffle operates on the AST level
  – Partial evaluation does not stop at DirectCallNode, but continues into next CallTarget
  – All later optimizations see the big combined tree, without further work

• Function splitting creates a new, uninitialized copy of an AST
  – Specialization in the context of a particular caller
  – Useful to avoid polymorphic specializations and to keep polymorphic inline caches shorter
  – Function inlining can inline a better specialized AST
  – Result: context sensitive profiling information

• Function inlining and function splitting are language independent
  – The Truffle framework is doing it automatically for you
Compilation with Inlined Function

**SL source code without call:**

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

**Machine code for loop without call:**

```assembly
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
ejl L1
... L3: call transferToInterpreter
```

**SL source code with call:**

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

**Machine code for loop with call:**

```assembly
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
ejl L1
... L3: call transferToInterpreter
```

Truffle gives you function inlining for free!
Truffle as an Internal DSL

• The base VM, Graal, is based on partial evaluation without code generation
  • Annotations are only used to denote fields that should be viewed as final by the partial evaluator

• Initial language implementations included a lot of boilerplate
  • E.g. multiple execute() methods that differed only in argument/return types
    • To specialize for types
    • Complicated, handwritten logic to choose and combine specializations

• Truffle DSL
  • Implemented purely within Java, using annotations
  • Annotation processor reads annotations, generates additional code

• Case study: partial JavaScript interpreter
  • 3500 LOC in Java $\rightarrow$ 1000 LOC in Java + Truffle annotations
  • Ran faster (more consistent optimizations) and less error-prone
Overall System Structure

- Interpreter for every language
- Common API separates language implementation, optimization system, and tools (debugger)
- Integrate with Java applications
- Low-footprint VM, also suitable for embedding
- Language agnostic dynamic compiler

Diagram:
- Common API separates
  - language implementation
  - optimization system
  - tools (debugger)
- Integrate with Java applications
- Low-footprint VM, also suitable for embedding
- Language agnostic dynamic compiler
Performance: Graal VM

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

Speedup, higher is better

<table>
<thead>
<tr>
<th>Language</th>
<th>Graal</th>
<th>Best Specialized Competition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>1.02</td>
<td>1.2</td>
</tr>
<tr>
<td>Scala</td>
<td>4.1</td>
<td>4.5</td>
</tr>
<tr>
<td>Ruby</td>
<td>4.5</td>
<td>0.85</td>
</tr>
<tr>
<td>R</td>
<td>0.9</td>
<td>0.85</td>
</tr>
<tr>
<td>Native</td>
<td>0.9</td>
<td>0.85</td>
</tr>
<tr>
<td>JavaScript</td>
<td>0.9</td>
<td>0.85</td>
</tr>
</tbody>
</table>
Demonstration
Tools
Tools: We Don’t Have It All
(Especially for Debuggers)

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

• Productivity tradeoffs for programmers
  – Performance – disabled optimizations
  – Functionality – inhibited language features
  – Complexity – language implementation requirements
  – Inconvenience – nonstandard context (debug flags)
Tools: We Can Have It All

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

• Generalized *instrumentation* support
  1. Access to execution state & events
  2. Minimal runtime overhead
  3. Reduced implementation effort (for languages *and* tools)
Implementation Effort: Language Implementors

• Treat AST syntax nodes specially
  – Precise source attribution
  – Enable probing
  – Ensure stability

• Add default tags, e.g., Statement, Call, ...
  – Sufficient for many tools
  – Can be extended, adjusted, or replaced dynamically by other tools

• Implement debugging support methods, e.g.
  – Eval a string in context of any stack frame
  – Display language-specific values, method names, ...

• More to be added to support new tools & services
“Mark Up” Important AST Nodes for Instrumentation

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

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

Tag: Statement
Access to Execution Events

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

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

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

ProbeNode
- Manages “instrument chain” dynamically
- Propagates Events
- Instrumentation Type
Building Debuggers and Other Tools: We Can “Have it All”
Position Paper ICOOOLPS ’15
Michael L. Van De Vanter
Oracle Labs
michael.van.de.vanter@oracle.com

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

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

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

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

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

```java
@Instrumentable(factory = SLStatementNodeWrapper.class)
public abstract class SLStatementNode extends Node {
    private boolean hasStatementTag;
    private boolean hasRootTag;

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

Annotation generates type-specialized WrapperNode

The set of tags is extensible, tools can provide new tags
Example: Debugger

mx repl

```plaintext
Example: Debugger

mx repl

Example: Debugger

```
NetBeans Debugger

• NetBeans has experimental Truffle debugging support

• Download latest nightly build of NetBeans
  – This demo uses nightly build 201606100002

• Install Truffle plugin
  – Tools -> Plugins -> Available Plugins -> search for "Truffle"
  – Install "Truffle Debugging Support"

• Start SL in debug mode
  – sl -debug tests/SumObject.sl

• Manually insert debugger; statement into SumObject.sl

• Attach NetBeans debugger to port 8000
Example: NetBeans Debugger

```
sl -debug tests/SumObject.sl
```

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

Stepping and Variables view work as expected

Stacktrace view has small rendering issues


• Interesting comparison: tracing-based metacompilation (e.g. PyPy)
  • Bolz et al. Tracing the meta-level: PyPy's tracing JIT compiler. ICOOLPS 2009.