Truffle

Virtual Machines and Execution Environments, WS2014/15

Jan Graichen, Fabio Niephaus, Matthias Springer, Malte Swart

Hasso Plattner Institute, Software Architecture Group

December 4, 2014

Handout only: Credits

This paper is based on the paper
One VM to Rule Them All [2] by Würthinger et al.

Overview

How to Implement a Programming Language?

1. Prototype: build an abstract syntax tree (AST) interpreter
   - Easy to implement
   - But slow (tree traversal, virtual method calls)
2. Make it fast
   - Build a VM
   - Compile AST to byte code
   - JIT compilation

   $\rightarrow$ Hard to implement, reinvent the wheel (memory management etc.)

Truffle – "How it should be":

Build a parser, define an AST and add language specific optimizations to make it fast.

Overview

Infrastructure [2]

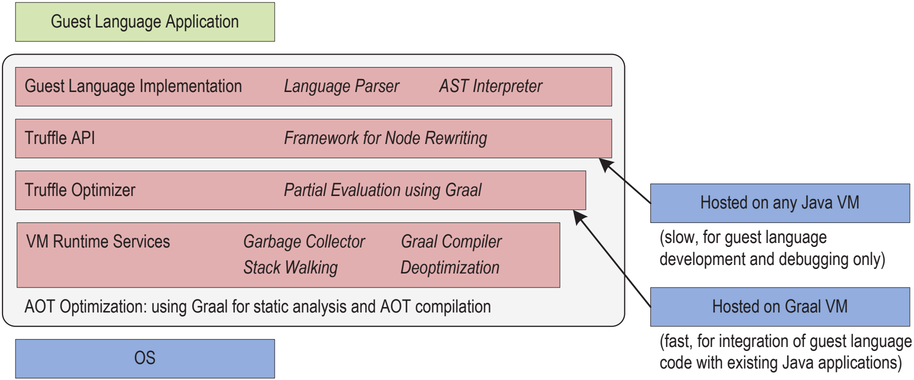


Figure: Interaction Graal/Truffle

Handout only: Infrastructure

- Main components
  - Truffle: provides guest language implementation API, support for optimization through node rewriting
  - Graal VM: HotSpot VM with Java API (instructured by Truffle)
- Two levels of optimization: Truffle, modified Graal VM

Truffle: How It Works

- Truffle: AST interpreter framework
- Framework to easily implement specialized nodes
- Based on AST node rewriting

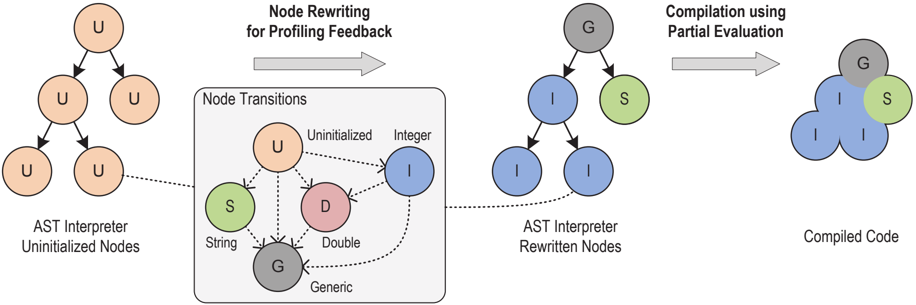
Sample Code (running example)

```
function showSumMilliseconds(a, b) {
 return (a + b) + " ms";
}
```

Code Example #1

```java
public Object add(...) {
  Object left = leftNode.executeGeneric(...);
  Object right = rightNode.executeGeneric(...);

  if (left instanceof Long && right instanceof Long) {
    try {
      return ExactMath.addExact((Long) left, (Long) right);
    } catch (ArithmeticException ex) { }
  }

  if (left instanceof Long)
      left = ((Long) left).doubleValue();
  if (right instanceof Long)
      right = ((Long) right).doubleValue();

  if (left instanceof Double && right instanceof Double)
    return (Double) left + (Double) right;

  if (left instanceof String || right instanceof String)
    return left.toString() + right.toString();

  throw new UnsupportedSpecializationException(...);
}
```

Node Rewriting [2]



Sample Code

```
function showSumMilliseconds(a, b) {
  return (a + b) + " ms";
}
showSumMilliseconds(1, 2);
```

Handout only: Node Rewriting

- Generic nodes can handle all types.
- Guards check if the type specialization is still accurate.
- Partial evaluation once a tree stablized (no rewrites for a while) and is *hot*.
  - Inlines `execute()` methods generates native code.
  - Adds a check and a deoptimization call where a rewrite could happen.
  - Requires Graal (accessing compiler with Java code).
- Truffle without Graal: interpreter mode

Deoptimization [2]

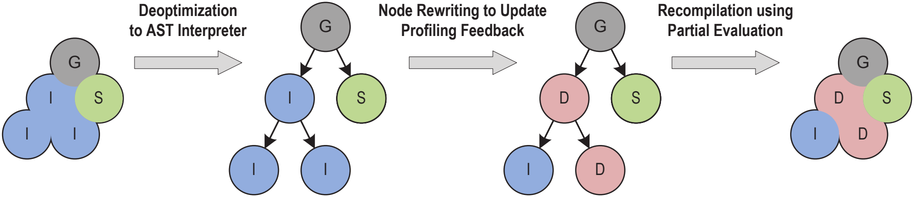


Figure: Deoptimization of Native Code

Sample Code

```
function showSumMilliseconds(a, b) {
  return (a + b) + " ms";
}
showSumMilliseconds(1, 2.5);
```

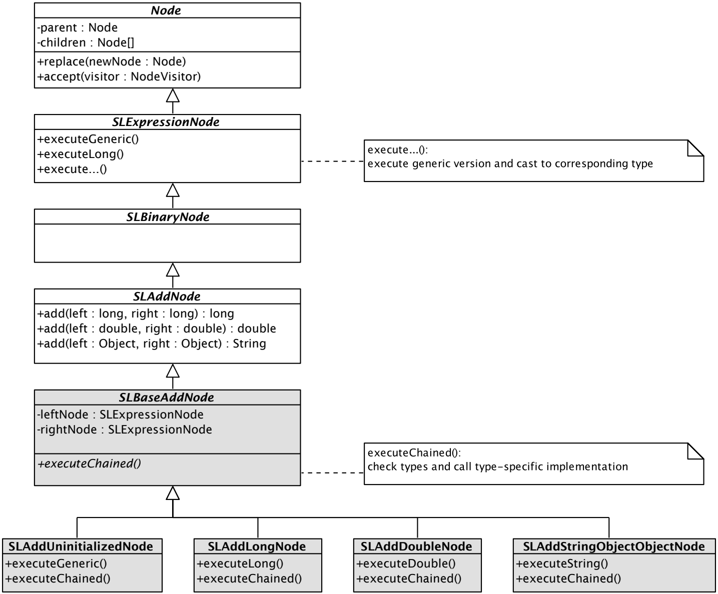
Handout only: Deoptimization

- Switch from compiled mode to interpreted mode if safety guard fails
- Reconstruction of program state in interpreter
- Node rewriting (see previous slides)
  - Switch from specialized node to generic node
  - In this example: switch from `integer` node to `double` node directly, because `double` nodes can also handle the `integer` case
- Partial evaluation (see previous slides)

Code Example #2 (using Annotation-Based DSL)

```
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 double add(double left, double right) {
    return left + 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;
  }
}
```

Classes Generated by DSL Preprocessor



Handout only: Code example

- Defined by language implementor:
  `SLExpressionNode, SLBinaryNode, SLAddNode`
- Generic case is generated by preprocessor
- **Uninitialized node:** replaces itself with specialized node
- **Monomorphic node:** one specialization only
- **Megamorphic node:** node can handle all types
  (last item on linked list)

Overview

Type Decision Chains [3]

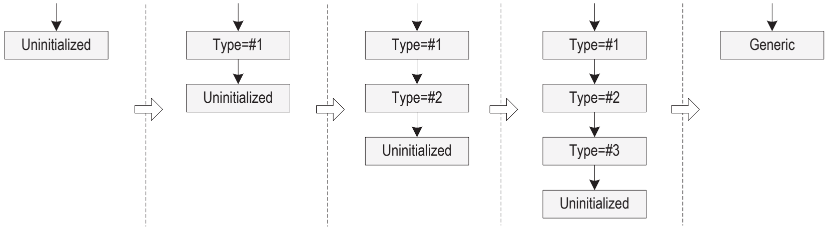
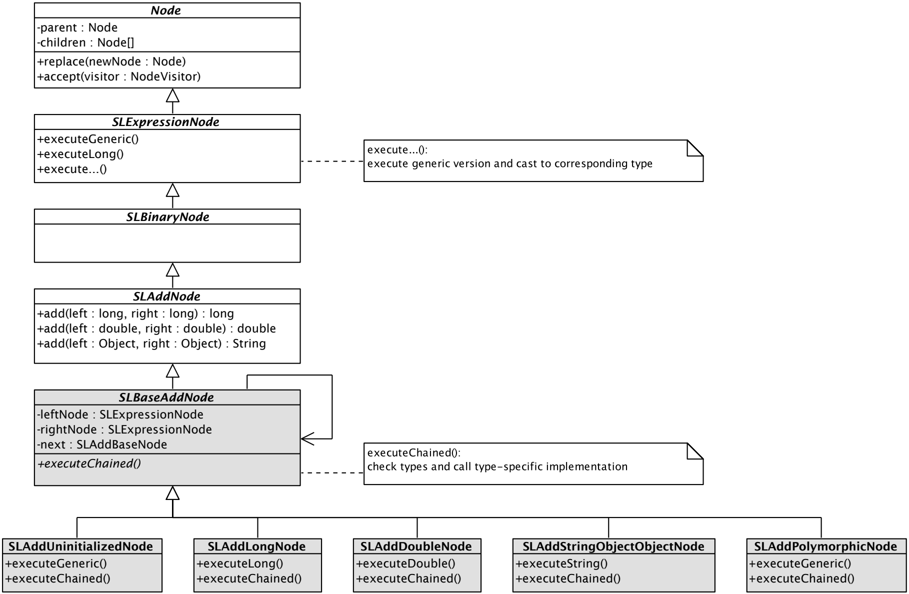


Figure: Type Decision Chains as Truffle's implementation of Polymorphic Inline Caches

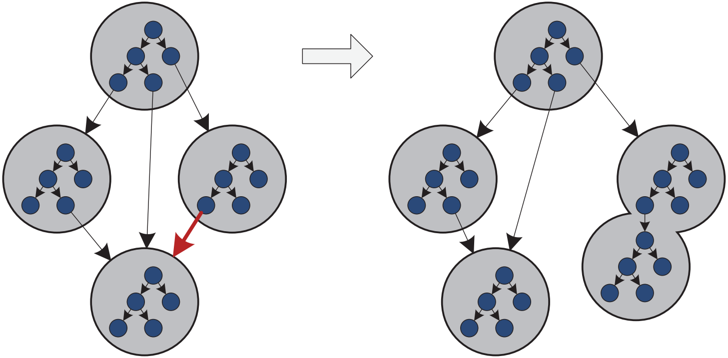
Classes Generated by DSL Preprocessor



Handout only: Type Decision Chains

- **Polymorphic node:** node can handle a limited set of types (linked list via `next` field): polymorphic inline caching
- Last element in linked list is megamorphic

AST Inlining [3]



Slightly More Complex Example

Sample Code

```
function foo() {
    return add(1, 2) + add("hello", "world");
}

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

AST Inlining [3]

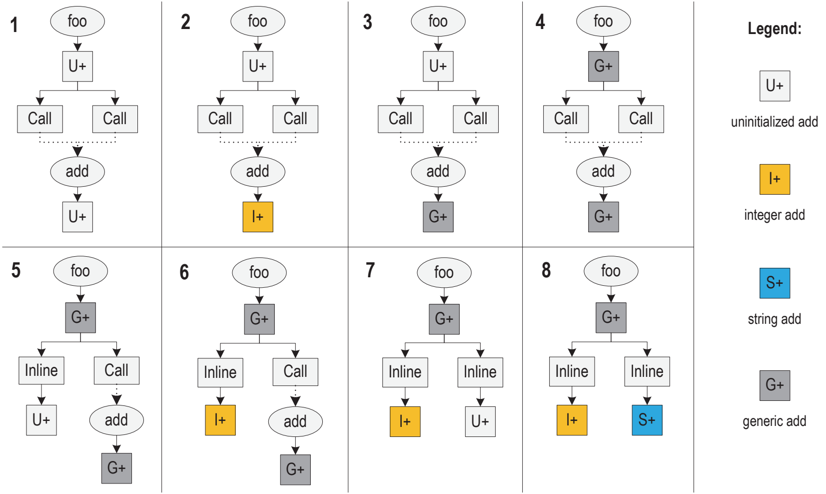


Figure: AST Evolution

Handout only: AST Inlining

- Duplicate parts of the AST.
- Every duplicate subtree can have its own specialization.

Assumptions

Requirement

Global assumptions about system state, like:

- Redefinition of system objects or methods (JavaScript, Ruby)
- Current class hierarchy (Java)

- Global one-time switch; bool can be changed to `false` once
- Partial evaluation with constant value instead of check
- On state change code is deoptimized

No runtime overhead in compiled code

Code Example #3: Assumption for Method Redefinition

```
final class MyCallNode {
  private final MyFunction function;
  private final Assumption functionStable;

  protected SLDirectDispatchNode(...MyFunction function) {
    this.function = function;
    this.functionStable = function.getStableAssumption();
  }

  protected Object execute(...) {
    try {
      functionStable.check();
      return function.call(...);
    } catch(InvalidAssumptionException ex) {
      replace(...);
    }
  }
}
```

Local Variables

Requirement

Highly efficient access to local variables while simple modeling

- Modeled as an array on `Frame` object
- Access nodes must be specializable for dynamic profiling

- Escape analysis of local variable array access
- Implicit single static assignment (SSA) form
- Host compiler can optimize without flow analysis
- `Frame` array never allocated except on deoptimization

As fast as host language variables; optional `Frame` facilities

Single Static Assignment (SSA) Form

Original Sample Code

```
if (condition) {
  x = value1 + value2;
} else {
  x = value2;
}
return x * 2;
```

Sample Code in SSA Form

```
if (condition) {
  x1 = value1 + value2;
} else {
  x2 = value2;
}
x3 = phi(x1, x2);
x4 = x3 * 2;
return x4;
```

Handout only: Single Static Assignment (SSA) Form

- Every variable is only written once.
- `phi` nodes capture variables from different branches.
- Replaces read access with address from last write.
- All variables are implicitly final/constant.
- Makes it easier to do certain optimizations (e.g. dead code elimination, common subexpression elimination, ...).

Overview

Languages

JavaScript

- Specialization of JavaScript generic types
- Object prototype chain changing by "shape" (assumptions)

Ruby

- Mostly method invocation $\rightarrow$ in-lining and shaping
- Method redefinitions via assumptions

Debugging (1/3)

Problem

Normal debugging:

- Different behavior when debugging: Disabled or different optimizations, different runtime behavior
- (Extremely) slower execution – may not practical to run production applications

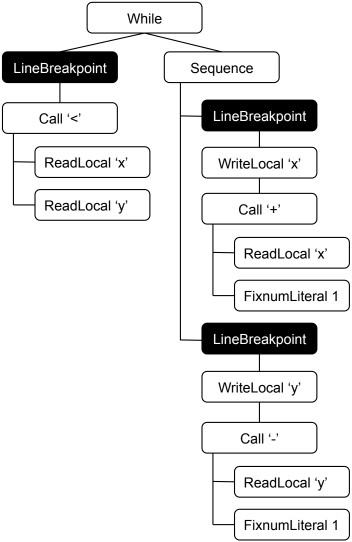
→ Use node rewriting and assumption for nearly zero overhead debugging

Debugging (2/3) [1]

Idea

Handle break points as simple AST nodes
Optimize using assumptions

```
while x < y
  x += 1
  y -= 1
end
```



Debugging (3/3): Results [1]

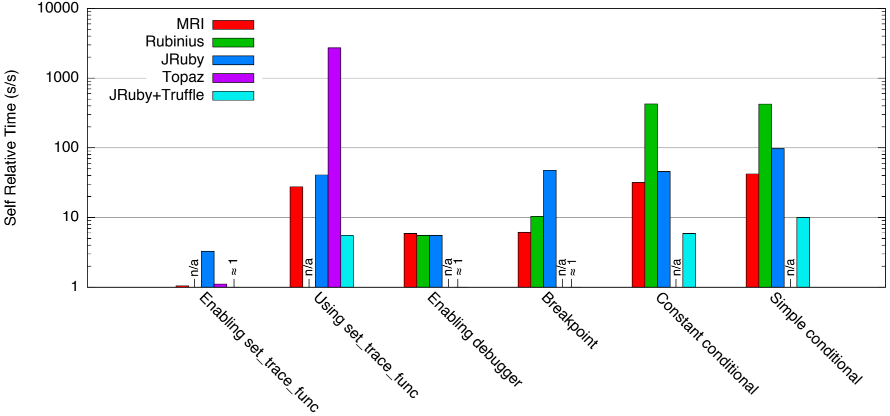


Figure: Relative debugging performance in different Ruby VM implementations

Overview

Summary

- Truffle is an AST interpreter framework
- Truffle lets developers concentrate on their domain, not having to implement generic optimizations again and again
- Truffle's powerful node rewriting technique supports most kinds of domain specific specialization
- Truffle therefore allows easy development of very fast AST interpreter

Future Work

We want to dive deeper and look at interesting stuff in:

- Interaction with Graal VM
- Partial Evaluation
- Deoptimization
- DSL Preprocessor
- Type System
- JRuby

Overview

References

1. C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at full speed. In Proceedings of the 8th Workshop on Dynamic Languages and Applications (DYLA), 2014.
2. T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, M. Wolczko. One VM to Rule Them All, 2013.
3. T. Würthinger, A. Woß, L. Stadler, G. Duboscq, D. Simon, C. Wimmer. Self-Optimizing AST Interpreters, 2012.