AST Specialisation and Partial Evaluation for Easy High-Performance Metaprogramming

1st Workshop on Meta-Programming Techniques and Reflection (META)

Chris Seaton
Research Manager
Oracle Labs
November 2016
Safe Harbor Statement

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

• We are using a novel combination of techniques to create high performance implementations of existing languages
  – Truffle: framework for writing AST interpreters in Java
  – Graal: new dynamic (JIT) compiler for the JVM that knows about Truffle
• We’ve found that this combination of tools is particularly useful for easy, pervasive, consistent, high-performance metaprogramming implementations
• We’ll show why this is and what it looks like
• We’ll suggest what properties from Truffle and Graal could be useful to make sure future language implementation systems have
Truffle and Graal
Truffle for AST interpreters
x + y * z

load_local x
load_local y
load_local z
call *
call +

pushq %rbp
movq %rsp, %rbp
movq %rdi, -8(%rbp)
movq %rsi, -16(%rbp)
movq %rdx, -24(%rbp)
movq -16(%rbp), %rax
movl %eax, %edx
movq -24(%rbp), %rax
imull %edx, %eax
movq -8(%rbp), %rdx
addl %edx, %eax
popq %rbp
ret
x + y * z

load_local x
load_local y
load_local z
call *
call +

pushq %rbp
movq %rsp, %rbp
movq %rdi, -8(%rbp)
movq %rsi, -16(%rbp)
movq %rdx, -24(%rbp)
movq -16(%rbp), %rax
movl %eax, %edx
movq -24(%rbp), %rax
imull %edx, %eax
movq -8(%rbp), %rdx
addl %edx, %eax
popq %rbp
ret
\[ x + y \times z \]

```
load_local x
load_local y
load_local z
call *
call +
```

```
pushq %rbp
movq %rsp, %rbp
movq %rdi, -8(%rbp)
movq %rsi, -16(%rbp)
movq %rdx, -24(%rbp)
movq -16(%rbp), %rax
movl %eax, %edx
movq -24(%rbp), %rax
imull %edx, %eax
movq -8(%rbp), %rdx
addl %edx, %eax
popq %rbp
ret
```
Node Rewriting for Profiling Feedback

AST Interpreter
Uninitialized Nodes

Node Rewriting for Profiling Feedback

Node Transitions

- Uninitialized
- Integer
- String
- Double
- Generic

AST Interpreter
Uninitialized Nodes

Graal for partial evaluation
Node Rewriting for Profiling Feedback

AST Interpreter
Rewritten Nodes

Compilation using Partial Evaluation

Compiled Code

Deoptimization to AST Interpreter

Node Rewriting to Update Profiling Feedback

Recompilation using Partial Evaluation

Metaprogramming in Ruby
# Conventional send
object.method_name(arg1, arg2, ...)

# Metaprogramming send
object.send('method_name', arg1, arg2, ...)
operator = exclude_end? ? <: : :<=
value.send(operator, last)
send("decode_png_resample_#{bit_depth}bit_value")
def method_missing(method, *args)
    @encapsulated_value.send(method, *args)
end
```ruby
def method_missing(name, *args)
  if Color.respond_to?(name)
    return Color.send(name, *args)
  end
end
end
```
eval(generated_template, variables)
object.instance_variable_get('@variable_name')
object.instance_variable_set('@variable_name', value)
def eql?(other)
    @hash.eql?(other.instance_variable_get(:@hash))
end
Foundational techniques
Caching

```ruby
a = [1, 2, 3]
puts a[2]

h = {1=>a, 2=>b, 3=>c}
puts h[2]
```

<table>
<thead>
<tr>
<th>Class</th>
<th>Method name</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>[]</td>
<td>Array#[]</td>
</tr>
<tr>
<td>Hash</td>
<td>[]</td>
<td>Hash#[]</td>
</tr>
</tbody>
</table>

... more entries ...

*one table per virtual machine, lots of entries*
Inline caching

```ruby
a = [1, 2, 3]
puts a[2]

h = {1=>a, 2=>b, 3=>c}
puts h[2]
```

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>Array#[]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>Hash#[]</td>
</tr>
</tbody>
</table>

*one table per call site, one entry*
Polymorphic inline caching

\[ x = \text{random}(a, h) \]
\[ x[2] \]

---

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>Array#[]</td>
</tr>
</tbody>
</table>

*one table per call site, one entry*

---

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>Hash#[]</td>
</tr>
</tbody>
</table>

*one table per call site, one entry*
Polymorphic inline caching

```latex
\begin{align*}
x &= \text{random}(a, h) \\
x[2] &
\end{align*}
```

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>Array#[]</td>
</tr>
<tr>
<td>Hash</td>
<td>Hash#[]</td>
</tr>
<tr>
<td>.... more entries ...</td>
<td></td>
</tr>
</tbody>
</table>

**one table per call site, multiple entries**

Dispatch chains

```
bit_depth = random(8, 16, 32)
send(image, "resample_#{bit_depth}bit")
```

<table>
<thead>
<tr>
<th>Class</th>
<th>Method name</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>resample_8bit</td>
<td>Image#resample_8bit</td>
</tr>
<tr>
<td>Image</td>
<td>resample_16bit</td>
<td>Image#resample_16bit</td>
</tr>
<tr>
<td>Image</td>
<td>resample_32bit</td>
<td>Image#resample_32bit</td>
</tr>
</tbody>
</table>

... more entries ...

*one table per call site, multiple entries*

Why aren’t these a solution on their own?
Caches are currently implemented manually

```c
struct rb_call_cache {
    /* inline cache: keys */
    rb_serial_t method_state;
    rb_serial_t class_serial;

    /* inline cache: values */
    const rb_callable_method_entry_t *me;

    vm_call_handler call;

    union {
        unsigned int index; /* used by ivar */
        enum method_missing_reason method_missing_reason; /* used by method_missing */
        int inc_sp; /* used by cfunc */
    } aux;
};
```
You need somewhere to store an inline cache

```
  a.foo(b)

  a = [1, 2, 3]
  a.sort

  a.send(:foo, b)
```
You need somewhere to store an inline cache

```ruby
a.foo(b)

a = [1, 2, 3]
a.sort

a.send(:foo, b)
```

\[ a \leftrightarrow b \]
You need somewhere to store an inline cache

```haskell
a.foo(b)
a = [1, 2, 3]
a.sort
a.send(:foo, b)
```
You need somewhere to store an inline cache

```java
@JRubyMethod(name = "send")
public static IRubyObject send(ThreadContext context, IRubyObject self, String name, IRubyObject[] args) {
    DynamicMethod method = searchMethod(name);
    return method.call(context, self, this, name, args);
}
```
You need somewhere to store an inline cache

```java
@JRubyMethod(name = "sort")
public static IRubyObject sort(ThreadContext context, IRubyObject array, String name) {
    ...
    Arrays.sort(newValues, 0, length, new Comparator() {
        public int compare(Object o1, Object o2) {
            DynamicMethod method = searchMethod("<=>");
            return method.call(context, self, this, name, o1, o2);
        }
    });
    ...
}
```
Caches quickly become megamorphic

send :foo  send :bar  send :baz

[:foo, :bar, :baz]
Caches quickly become megamorphic

\[ [a, b, c, d] \]
How Truffle and Graal make a difference
An easy place to store state

class SendNode extends Node {
    String methodName;
    Node receiverNode;

    public Object execute() {
        Object receiver = receiverNode.execute();
        Method method = receiver.lookup(methodName);
        return method.call();
    }
}
An easy place to store state

class SendNode extends Node {
    String methodName;
    Node receiverNode;
    Class cachedClass;
    Method cachedMethod;

    public Object execute() {
        Object receiver = receiverNode.execute();
        if (receiver.getClass() != cachedClass) {
            cachedClass = receiver.getClass();
            cachedMethod = receiver.lookup(methodName);
        }
        return cachedMethod.call();
    }
}
A DSL to write caches in just a couple of lines

```java
@NodeChild("receiver")
class SendNode extends Node {
    String methodName;

    @Specialisation(guards = "receiver.getClass() == cachedClass")
    public Object execute(Object receiver,
                           @Cached("receiver.getClass()") Class cachedClass,
                           @Cached("receiver.lookup(methodName)") Method cachedMethod) {
        return method.call();
    }
}
```
A DSL to write caches in just a couple of lines

```java
@NodeChildren({"receiver", "name"})
class SendNode extends Node {
    @Specialisation(guards = {"receiver.getClass() == cachedClass", "name.equals(cachedName)"})
    public Object execute(Object receiver,
        String name,
        @Cached("receiver.getClass()") Class cachedClass,
        @Cached("name") String cachedName,
        @Cached("receiver.lookup(name)") Method cachedMethod) {
        return method.call();
    }
}
```
Automatic splitting to push caches down the call stack

\[ [a, b, c, d] \]
Automatic splitting to push caches down the call stack
Automatic splitting to push caches down the call stack
Results
def eql?(other)
    @hash.eql?(other.instance_variable_get(:@hash))
end
Relative performance of metaprogramming access to instance variables relative to conventional access
Slowdown of metaprogramming access to instance variables relative to JRuby+Truffle
Slowdown of `Set#eq1?` relative to JRuby+Truffle
The important properties
Somewhere to store state

• Caching and profiling requires somewhere to store state
• Truffle’s nodes are just Java objects, so you can store whatever you want in normal Java fields
• In Truffle you are almost always in a node, so you almost always have access to your state
  – Doesn’t become inaccessible in compiled code
Low-effort caching

• Truffle’s DSL makes it easy to add sophisticated polymorphic inline caches anywhere

• This is implemented using the state that we just mentioned

• Guards can be arbitrary Java expressions, or zero-overhead mutable flags using deoptimisation

• Supports an arbitrary number of guards
Dynamic optimisation

• Dynamic optimisation (JIT compilation) comes for free from Graal
• Partial evaluation removes degrees of freedom that aren’t used
  – Allows us to add degrees of freedom to handle metaprogramming without worrying
Dynamic deoptimisation

• Allows us to make speculative optimisations and reverse them if they were wrong
• Allows functionality not used to be ‘turned off’ until it is needed
• Allows local variables to be lowered all the way to registers while still letting frames be accessed as if they were objects
Automatic inlining and splitting

• Removes the overhead of intermediate methods calls and indirection used in metaprogramming

• Allows state to be ‘pushed down’ the call stack to reduce polymorphism
Programmatic access to frames

• Allows local variables to be read and written from outside method activations
• Whole frames represented as objects
• Access to the list of frames currently on the stack
Conclusions

• We already knew how to make most (not all) of Ruby’s metaprogramming functionality fast
• Existing mature Ruby implementations don’t apply this knowledge
• Why? Because it was hard in practice to do it consistently and pervasively that they never got around to it
Conclusions

• Truffle and Graal make it so much easier
• We’ve identified what we think are the key properties that enable this
• I think Truffle and Graal are the only systems to provide effective implementations of these
• If you are implementing a metaprogramming language, use Truffle and Graal
• If you’re making a new language implementation system, perhaps incorporate these same properties
Where to find more information
Truffle and Graal: Fast Programming Languages With Modest Effort

Thursday, 14:20, Matterhorn 3 (this room)
SPLASH-I
Adam Welc
Acknowledgements

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

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

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

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

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

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

LaBRI
Floréal Morandat

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

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

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

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

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

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