JRuby 9000
Optimizing Above the JVM
Me

• Charles Oliver Nutter (@headius)
• Red Hat
• Based in Minneapolis, Minnesota
• Ten years working on JRuby (uff da!)
Ruby Challenges

- Dynamic dispatch for most things
- Dynamic possibly-mutating constants
- Fixnum to Bignum promotion
- Literals for arrays, hashes: `[a, b, c].sort[1]`
- Stack access via closures, bindings
- Rich inheritance model
module SayHello
  def say_hello
    "Hello, " + to_s
  end
end

class Foo
  include SayHello
  
  def initialize
    @my_data = {bar: 'baz', quux: 'widget'}
  end

  def to_s
    @my_data.map do |k,v|
      "#{k} = #{v}"
    end.join(', ')
  end
end

Foo.new.say_hello # => "Hello, bar = baz, quux = widget"
More Challenges

• "Everything's an object"
• Tracing and debugging APIs
• Pervasive use of closures
• Mutable literal strings
JRuby 9000

• Optimizable intermediate representation
• Mixed mode runtime (now with tiers!)
• Lazy JIT to JVM bytecode
• byte[] strings and regular expressions
• Lots of native integration via FFI
• 9.0.5.0 is current
Intermediate Representation

- AST to semantic representation
- Traditional compiler design
- Register machine
Lexical Analysis

Parsing

AST

Interpret

Bytecode Generation

Semantic Analysis

Optimization

IR Instructions

CFG

DFG

...


Register-based

def foo(a, b)
c = 1
d = a + c
end

IR Instructions
0  check_arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
2  b = recv_pre_reqd_arg(1)
3  %block = recv_closure
4  thread_poll
5  line_num(1)
6  c = 1
7  line_num(2)
8  %v_0 = call(:+, a, [c])
9  d = copy(%v_0)
10 return(%v_0)

3 address format
def foo(a, b):
    c = 1
    d = a + c
end

-Xir.passes=LocalOptimizationPass, DeadCodeElimination

0  check arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
2  b = recv_pre_reqd_arg(1)
3  %block = recv_closure
4  thread_poll
5  line_num(1)
6  c = 1
7  line_num(2)
8  %v_0 = call(:+, a, [c])
9  d = copy(%v_0)
10 return(%v_0)
def foo(a, b)
    c = 1
    d = a + c
end

-Xir.passes=LocalOptimizationPass,
DeadCodeElimination

0  check arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
2  b = recv_pre_reqd_arg(1)
3  %block = recv_closure
4  thread_poll
5  line_num(1)
6  c = 1
7  line_num(2)
8  %v_0 = call(:+, a, [c])
9  d = copy(%v_0)
10 return(%v_0)
def foo(a, b)
    c = 1
    d = a + c
end

-Xir.passes=LocalOptimizationPass, DeadCodeElimination

0  check_arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
2  b = recv_pre_reqd_arg(1)
3  %block = recv_closure
4  thread_poll
5  line_num(1)
6  c = 1
7  line_num(2)
8  %v_0 = call(:+, a, [c])
9  d = copy(%v_0)
10 return(%v_0)
```python
def foo(a, b):
c = 1

d = a + c
end
```

```assembly
0  check_arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
4  thread_poll
5  line_num(1)
6  c = 1
7  line_num(2)
8  %v_0 = call(:+, a, [c])
9  d = copy(%v_0)
10 return(%v_0)
```
```
def foo(a, b)
c = 1
d = a + c
end
```

```
0  check_arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
4  thread_poll
5  line_num(1)
6  c = 1
7  line_num(2)
8  %v_0 = call(:+, a, [c])
9  d = copy(%v_0)
10 return(%v_0)
```
def foo(a, b):
    c = 1
    d = a + c
end

-Xir.passes=LocalOptimizationPass, DeadCodeElimination

0  check_arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
4  thread_poll
5  line_num(1)
6  c = 1
7  line_num(2)
8  %v_0 = call(:+, a, [c])
9  d = copy(%v_0)
10 return(%v_0)
def foo(a, b):
c = 1
d = a + c
end

0  check arity(2, 0, -1)
1  a = recv pre reqd arg(0)
4  thread poll
5  line num(1)
6  c =
7  line num(2)
8  %v_0 = call(+:, a, [1])
9  d = copy(%v_0)
10 return(%v_0)
def foo(a, b):
    c = 1
    d = a + c
end

-Xir.passes=LocalOptimizationPass, DeadCodeElimination

0  check arity(2, 0, -1)
1  a = recv pre reqd arg(0)
4  thread poll
5  line num(1)
7  line num(2)
8  %v_0 = call(:+, a, [1])
9  d = copy(%v_0)
10 return(%v_0)
0  check arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
4  thread_poll
5  line_num(1)
7  line_num(2)
8  %v_0 = call(:+, a, [1])
9  d = copy(%v_0)
10 return(%v_0)
check_arity(2, 0, -1)

a = recv_pre_reqd_arg(0)

thread_poll

line_num(2)

%v_0 = call(:+, a, [1])

d = copy(%v_0)

return(%v_0)
Tiers!

- Tier 1: Simple interpreter (no passes run)
- Tier 2: Full interpreter (static optimization)
- Tier 3: Full interpreter (profiled optz)
- Tier 4: JVM bytecode (static)
- Tier 5: JVM bytecode (profiled)
- Tiers 6+: Whatever JVM does from there
Why Not Truffle?

- Startup and memory use are worse
- No integration with other JVM langs yet
- We still want to target JVM
- It's not ready yet!
Current Optimizations
Block Jitting

- JRuby 1.7 only jitted methods
- Not free-standing procs/lambdas
- Not define_method blocks
- Easier to do now with 9000's IR
- Blocks JIT as of 9.0.4.0
define_method

```ruby
define_method(:add) do |a, b|
  a + b
end

names.each do |name|
  define_method(name) { send :"do_#{name}" }
end
```

Convenient for metaprogramming, but blocks have more overhead than methods.
Optimizing define_method

- Noncapturing
  - Treat as method in compiler
  - Ignore surrounding scope
- Capturing (future work)
  - Lift read-only variables as constant
Getting Better!

- def
- define_method
- define_method w/ capture

- MRI
- JRuby 9.0.1.0
- JRuby 9.0.4.0
Low-cost Exceptions

- Backtrace cost is VERY high on JVM
- Lots of work to construct
- Exceptions frequently ignored
- ...or used as flow control (shame!)
- If ignored, backtrace is not needed!
Postfix Antipattern

foo rescue nil

Exception raised
StandardError rescued
Exception ignored

Result is simple expression, so exception is never visible.
csv.rb Converters

Converters = { integer: lambda { |f| 
    Integer(f) rescue f 
  },
  float:     lambda { |f| 
    Float(f) rescue f 
  },
  ...

All trivial rescues, no traces needed.
Strategy

• Inspect rescue block
• If simple expression...
  • Thread-local requiresBacktrace = false
  • Backtrace generation short circuited
• Reset to true on exit or nontrivial rescue
public class ToggleBacktraceInstr extends NoOperandInstr {
    private final boolean requiresBacktrace;

    public ToggleBacktraceInstr(boolean requiresBacktrace) {
        super(Operation.TOGGLE_BACKTRACE);

        this.requiresBacktrace = requiresBacktrace;
    }
}
Simple rescue Improvement

Iters/second

524,475

10,700
Much Better!

![Bar chart comparing iters/second]

- 10,700
- 524,475
Work In Progress
Object Shaping

- Ruby instance vars allocated dynamically
- JRuby currently grows an array
- We have code to specialize as fields
  - Working, tested
  - Probably next release
Inlining

- 900 pound gorilla of optimization
- shove method/closure back to callsite
- specialize closure-receiving methods
- eliminate call protocol
- We know Ruby better than the JVM
But... JVM?

- JVM will inline for us, but...
  - only if we use invokedynamic
  - and the code isn't too big
  - and there's no polymorphic code
  - and we're not yielding to a closure
  - and it feels like it today
def decrement_one(i)
    i - 1
end

i = 1_000_000
while i > 0
    i = decrement_one(i)
end

def decrement_one(i)
    i - 1
end

i = 1_000_000
while i < 0
    if guard_same? self
        i = i - 1
    else
        i = decrement_one(i)
    end
end
def decrement_one(i)
    i - 1
end

i = 1_000_000
while i > 0
    i = decrement_one(i)
end

def decrement_one(i)
    i - 1
end

i = 1_000_000
while i < 0
    if guard_same? self
        i = i - 1
    end
    else
        i = decrement_one(i)
    end
end
Today's Inliner

```python
def decrement_one(i):
    i -= 1
end

i = 1_000_000
while i > 0
    i = decrement_one(i)
end
```
Today's Inliner

```python
def decrement_one(i):
    i - 1
end

i = 1_000_000
while i > 0:
    i = decrement_one(i)
end
```

```python
def decrement_one(i):
    i - 1
end

i = 1_000_000
while i < 0:
    if guard_same? self
        i = i - 1
    else
        i = decrement_one(i)
end
end```
Profiling

• You can't inline if you can't profile!
• For each call site record call info
  • Which method(s) called
  • How frequently
• Inline most frequently-called method
Inlining a Closure

def small_loop(i):
    k = 10
    while k > 0
        k = yield(k)
    end
    i - 1
end

def big_loop(i):
    i = 100_000
    while true
        i = small_loop(i) { |j| j - 1 }
        return 0 if i < 0
    end
end

900.times { |i| big_loop i }
Inlining FTW!

![Bar chart showing time in seconds]

- **56.9** seconds
- **14.1** seconds
Profiling

- <2% overhead (to be reduced more)
- Working* (interpreter AND JIT)
- Feeds directly into inlining
- Deopt coming soon

* Fragile and buggy!
Interpreter FTW!

- Deopt is much simpler with interpreter
- Collect local vars, instruction index
- Raise exception to interpreter, keep going
- Much cheaper than resuming bytecode
Numeric Specialization

- "Unboxing"
- Ruby: everything's an object
  - Tagged pointer for Fixnum, Float
- JVM: references OR primitives
- Need to optimize numerics as primitive
But... EA?

- Hotspot will eliminate boxes if...
  - All code inlines
  - No branches in the code
  - Dynamic calls have type guards
- EA does not work for us!
def looper(n)
    i = 0
    while i < n
        do_something(i)
        i += 1
    end
end

Specialize n, i to long

def looper(long n)
    long i = 0
    while i < n
        do_something(i)
        i += 1
    end
end

deopt to object version if n or i + 1 is not Fixnum
Unboxing Today

- Working prototype
- No deopt
- No type guards
- No overflow check for Fixnum/Bignum
def iterate(x, y):
    cr = y - 0.5
    ci = x
    zi = 0.0
    zr = 0.0
    i = 0
    bailout = 16.0
    max_iterations = 1000

    while true
        i += 1
        temp = zr * zi
        zr2 = zr * zr
        zi2 = zi * zi
        zr = zr2 - zi2 + cr
        zi = temp + temp + ci
        return i if (zi2 + zr2 > bailout)
        return 0 if (i > max_iterations)
    end
end
Mandelbrot performance

- JRuby
- JRuby + truffle
Mandelbrot performance

- JRuby
- JRuby + truffle
- JRuby on Graal
When?

- Object shape should be in 9.1
- Profiling, inlining mostly need testing
- Specialization needs guards, deopt
- Probably landing in next couple months
Thank You

- Charles Oliver Nutter
- @headius
- headius@headius.com