Overview

Got a language cooking on the JVM?

JSR 292, a set of major changes to the JVM architecture, provides you with some exciting new ingredients.
Agenda

> A Discourse on Methods
  > discussion of compiled code
> Recipes (= use cases):
  > calling Java
  > Curry
  > Fast-and-slow
> (…with JSR 292 API elements sprinkled in)
What’s in a method call?
What’s in a method call?

- Naming — using a symbolic name
- Linking — reaching out somewhere else
- Selecting — deciding which one to call
- Adapting — agreeing on calling conventions
What’s in a method call?

> Naming — using a symbolic name
> Linking — reaching out somewhere else
> Selecting — deciding which one to call
> Adapting — agreeing on calling conventions

> (...and finally, a parameterized control transfer)
A connection from caller A to target B

- Including naming, linking, selecting, adapting:
- …where B might be known to A only by a name
- …and A and B might be far apart
- …and B might depend on arguments passed by A
- …and a correct call to B might require adaptations
A connection from caller A to target B

- Including naming, linking, selecting, adapting:
- ...where B might be known to A only by a name
- ...and A and B might be far apart
- ...and B might depend on arguments passed by A
- ...and a correct call to B might require adaptations

> (After everything is decided, A jumps to B’s code.)
Example: Fully static invocation

> For this source code

```java
String s = System.getProperty("java.home");
```

The compiled byte code looks like

```java
0: ldc #2           //String "java.home"
2: invokestatic #3  //Method java/lang/System.getProperty:
                (Ljava/lang/String;)Ljava/lang/String;
5: astore_1
```
Example: Fully static invocation

> For this source code

```java
String s = System.getProperty("java.home");
```

The compiled byte code looks like

```
0:  ldc #2           //String "java.home"
2:  invokestatic #3 //Method java/lang/System.getProperty:
      (Ljava/lang/String;)Ljava/lang/String;
5:   astore_1
```

a) Names are embedded in the bytecode
b) Linking handled by the JVM with fixed Java rules
c) Target method selection is not dynamic at all
d) No adaptation: Signatures must match exactly
How the VM sees it:

(Note: This implementation is typical; VMs vary.)
How the VM sees it:

(Note: This implementation is typical; VMs vary.)
How the VM sees it:

(Note: This implementation is typical; VMs vary.)
Example: Class-based single dispatch

For this source code

```java
//PrintStream out = System.out;
out.println("Hello World");
```

The compiled byte code looks like

```
4:   aload_1
5:   ldc #2            //String "Hello World"
7:   invokevirtual #4   //Method java/io/PrintStream.println:
                 (Ljava/lang/String;)V
```
Example: Class-based single dispatch

For this source code

```java
//PrintStream out = System.out;
out.println("Hello World");
```

The compiled byte code looks like

```
4:  aload_1
5:  ldc #2            //String "Hello World"
7:  invokevirtual #4  //Method java/io/PrintStream.println:
           (Ljava/lang/String;)V
```

a) Again, names in bytecode
b) Again, linking fixed by JVM

c) Only the receiver type determines method selection
d) Only the receiver type can be adapted (narrowed)
How the VM selects the target method:

(Note: This implementation is typical; VMs vary.)
How the VM selects the target method:

(Note: This implementation is typical; VMs vary.)
How the VM selects the target method:

```java
... 
aload 1
ldc "Hello World"
invvirt PrintStr
 .println(String)
... 
Constant Pool
... 
PrintStr.println = virtual[9]
...

1. load object header

2. load vtable entry

class PrintStr
vtable:
[8] &close
[9] &println
...

close():
...
println(s):
...
```

(Note: This implementation is typical; VMs vary.)
How the VM selects the target method:

(Note: This implementation is typical; VMs vary.)
How the VM selects the target method:

(Note: This implementation is typical; VMs vary.)
Dynamic method invocation

For this source code

```java
//Object x; Integer y;
if (InvokeDynamic.<boolean>lessThan(x, y))
```

A new option:

```java
0:   aload_1; aload_2
2:   invokedynamic #3 //NameAndType lessThan:
          (Ljava/lang/Object;Ljava/lang/Integer;)Z
5:   if_icmpeq
```

Advantages:

- Compact representation
- Local argument & return types recorded accurately
- (Flexibility from signature polymorphism.)
How the VM finds the target method:

```java
...  
aload_1; aload_2
invdyn  lessThan:Z
if_icmpeq
...
```

This pointer links to the target method, a "Method Handle."

```java
class Runtime
  lessThan(, Z):
  ...
```
The target method can be a chain:

```java
aload_1;aload_2
invdyn lessThan:Z
if_icmpeq
...
```

```
class Runtime

invoke_2(String message, Object, Object):
...
```

- `this chain of targets converts a return value to boolean, and inserts an extra message argument`
- `toBoolean Adapter`
- `Bound MH`
- `String "lessThan"`
- `direct MH`
invokedynamic bootstrap logic:

```
...  
aload_1; aload_2  
invdyn lessThan:Z  
if_icmpeq  
...  
```

the containing class must declare a bootstrap method to initialize its call sites on demand

```
class Runtime
bootstrap(info...):
...  
return new CallSite(info)
```
Method handles

> An object of static type `java.dyn.MethodHandle`
> Like methods, can have any function type
> Unlike (other) objects, signature-polymorphic
> Like methods, can be virtual, static, or “special”
> Unlike methods, not named
> Invoked like methods:
  ```java
  MethodHandle.invoke(args)
  ```
An invokedynamic call site

- An invokedynamic call site contains
  - A method signature (immutable)
  - A method name (arbitrary)
  - The enclosing caller class
  - A class-specific bootstrap method
  - A site-specific target method *(the payload!)*
  - A CallSite which reifies it all

- All immutable, except for target method
An invokedynamic call site (target)

> The linkage state consists only of the current target
> Target is a *method handle*
>   > May point directly to a Java method
>   > Can optionally test or adjust arguments
> Mutable property of the instruction
>   > (May be managed via a reified `CallSite` object)
>   > May be set at any time, but few changes expected
>   > Changing a target *may* affect compilation, etc.
Bootstrap methods

> The per-class “plug in” is the *bootstrap method*

> Its job is to build a reified call site on first execution
  > We consult the bootstrap *once*,
  > And then it gets out of the way

> Call site must have call-ready target from the start
  > target can be eagerly or lazily linked
  > can be a method handle for an inline cache
  > …can re-link the call site if prediction fails
An invokedynamic call site

> An invokedynamic call site contains
  ● A method signature (immutable)
  ● A method name (arbitrary)
  ● The enclosing caller class
  ● A class-specific bootstrap method
  ● A site-specific target method (*the payload!*)
  ● A CallSite which reifies it all

> All immutable, except for target method
An invokedynamic call site (target)

> The linkage state consists only of the current target
> Target is a method handle
>   > May point directly to a Java method
>   > Can optionally test or adjust arguments
> Mutable property of the instruction
>   > (May be managed via a reified CallSite object)
>   > May be set at any time, but few changes expected
>   > Changing a target may affect compilation, etc.
Let’s talk about compiled code
A Simple Ruby method

> For this source code

```ruby
def myadd(a, b)
    return a + b
end
```

consider the untyped plus “+” operation…
Not-so-simple compiled code

- The JVM compiles and inlines these methods:
The JVM compiles and inlines these methods:

```java
@ 25  test::method__2$RUBY$myadd inline (hot)
@ 1   org.jruby.runtime.ThreadContext::getRuntime inline (hot)
@ 7   org.jruby.Ruby::getNil inline (hot)
@ 22  test::setPosition inline (hot)
@ 4   org.jruby.runtime.ThreadContext::setFileAndLine inline (hot)
@ 26  org.jruby.ast.executable.AbstractScript::getCallSite inline (hot)
@ 35  org.jruby.runtimecallsite.CachingCallSite::call inline (hot)
    test::method__2$RUBY$myadd -> @ 35  org.jruby.runtime.callsite.CachingCallSite::call
>> TypeProfile (6700/6700 counts) = org/jruby/runtime/callsite/NormalCachingCallSite (54 bytes)
    @ 2   org.jruby.runtime.callsite.CachingCallSite::pollAndGetClass inline (hot)
    @ 1   org.jruby.ThreadContext::callThreadPoll inline (hot)
        @ 19  org.jruby.runtime.ThreadContext::pollThreadEvents executed <
MinInliningThreshold times
    @ 5   org.jruby.RubyBasicObject::getMetaClass inline (hot)
    @ 5   org.jruby.RubyBasicObject::getMetaClass inline (hot)
    org.jruby.runtime.callsite.CachingCallSite::pollAndGetClass -> @ 5
    org.jruby.RubyBasicObject::getMetaClass >> TypeProfile (2234/6701 counts) = org/jruby/RubyObject (5 bytes)
    org.jruby.runtime.callsite.CachingCallSite::pollAndGetClass -> @ 5
    org.jruby.RubyBasicObject::getMetaClass >> TypeProfile (4467/6701 counts) = org/jruby/RubyFixnum (5 bytes)
    @ 17  org.jruby.runtime.callsite.CacheEntry::typeOk inline (hot)
        @ 5   org.jruby.RubyModule::getCacheToken inline (hot)
        @ 38  org.jruby.RubyFixnum$.i_method_1$0$RUBYINVOKER$op_plus::call inline (hot)
    org.jruby.runtime.callsite.CachingCallSite::call -> @ 38  org.jruby.RubyFixnum$.i_method_1$0$RUBYINVOKER$op_plus::call
        @ 20  org.jruby.RubyFixnum$.i_method_1$0$RUBYINVOKER$op_plus (11 bytes)
    @ 7   org.jruby.RubyFixnum::op_plus inline (hot)
        @ 13  org.jruby.RubyFixnum::addFixnum inlining too deep
        @ 20  org.jruby.RubyFixnum::addOther too big
```
Not-so-simple compiled code

The JVM compiles and inlines these methods:

```java
@ 38  org.jruby.RubyFixnum$i_method_1_0$RUBYINVOKER
    $op_plus::call inline (hot)
org.jruby.runtime.callsite.CachingCallSite::call
-> @ 38  org.jruby.RubyFixnum$i_method_1_0$RUBYINVOKER
    $op_plus::call >>TypeProfile (6701/6701 counts) = org/jruby/
RubyFixnum$i_method_1_0$RUBYINVOKER$op_plus (11 bytes)
   @ 7  org.jruby.RubyFixnum::op_plus inline (hot)
   @ 13 org.jruby.RubyFixnum::addFixnum
```
inlining too deep
After optimization, optimistic type checks
After optimization, optimistic type checks

```
4f2   B81: #   B230 B82 <- B80  Freq: 0.999948
4f2   MOV   EBX,[EDI + #8] ! Field org/jruby/runtime/CallSite.methodName
4f5   MOV   [ESP + #44],EBX
4f9   MOV   EBP,[EDX + #8] ! Field org/jruby/runtime/callsite/CacheEntry.method
4fc   MOV   EBX,[EBP + #4]
4ff   NullCheck EBP
4ff   B82: #   B165 B83 <- B81  Freq: 0.999947
4ff   CMPu  EBX,precise klass org/jruby/RubyFixnum$i_method_1__0$RUBYINVOKER$op_plus:
      0x2b8ef050:Constant:exact *
      505    Jne,u  B165  P=0.000001 C=-1.000000
      505
      50b  B83: #   B202 B84 <- B82  Freq: 0.999946
      50b  CMPu  ECX,precise klass org/jruby/RubyFixnum: 0x2ba9eb58:Constant:exact *
      511    Jne,u  B202  P=0.000000 C=-1.000000
      511
      517  B84: #   B268 B85 <- B83  Freq: 0.999946
<here comes the add>
```
So, what can indy do?

> Currently only interpreted invokedynamic supported
>   It's 5 to 25% slower than “normal” Jruby
>   Compiled invokedynamic is almost there

> but there are still some issues (we are currently working on that)
JRuby is very smart!

- Generated “invoker” methods are inlined perfectly
- but you have to generate them
- these are a lot of bytecodes
  - Makes your implementation complex
  - Default inlining depth can be (and is) hit
  - Linear dispatching pattern hidden in call tree (?)
MethodHandles does that for you

- You get the speed of JRuby out-of-the-box
- Your language implementation is much simpler
  - you can concentrate on other things
- Compiled invokedynamic is very likely to have the same performance as JRuby's invoker methods
  - (but maybe some other compiler optimizations kick in that we currently don't think about)
- method handle chains are a clear signal of linear control flow to the inliner
Some code examples…
Plain old Java

```java
java.io.File file = new java.io.File("muffin.txt");
println(file.getName());
MethodHandle getName =
    LOOKUP.findVirtual(file.getClass(), "getName",
    MethodType.make(String.class));
println(getName.<String>invoke(file));
```

Method handles can access any method in any Java API.
Plain old Java

```java
MethodHandle charAt =
   LOOKUP.findVirtual(String.class, "charAt",
   MethodType.make(char.class, int.class));
println(charAt.<char>invoke("foam", 3));
```

Primitive types (like int, char) work just fine.
Plain old Java

```java
// invokedynamic
println(InvokeDynamic.<String>getName(file));
println(InvokeDynamic.<String>toString((CharSequence) "soy latte"));
println(InvokeDynamic.<String>
    #"static:\=java\\lang\\Integer:toHexString"
    (0xCAFE));
```

Invokedynamic sites can be bound to Java methods.
Curry (chicken or rice)

```java
MethodHandle list2 = Utensil.list(2);
println(list2);  // list2 = {(x,y) => Arrays.asList(x,y)}
println:invoke(list2, "chicken", "rice");  // [chicken, rice]

// curry with chicken or rice:
MethodHandle partialApp = insertArguments(list2, 0, "curry");
println(partialApp);  // partialApp = {x => list2("curry", x)}
println:invoke(partialApp, "chicken");  // [curry, chicken]
println:invoke(partialApp, "rice");  // [curry, rice]
```
Curry (with everything)

// curry with everything:
MethodHandle list3 = Utensil.list(3);
println(list3);  // list3 = {(x,y,z) => Arrays.asList(x,y,z)}
MethodHandle partialApp2 = insertArguments(list3, 0, "curry");  // partialApp2 = {(x, y) => list3("curry", x, y)}
println(partialApp2);
println:invoke(partialApp2, "chicken", "rice");  // [curry, chicken, rice]
Curry (in cascade)

```
// double curry:
MethodHandle pa3 = insertArguments(list3, 0, "curry", "chutney");
// pa3 = {x => list3("curry", "chutney", x)}
println(pa3);
println:invoke(pa3, "tofu"); //= [curry, chutney, tofu]

// triple curry:
MethodHandle pa4 = insertArguments(pa3, 0, "yak");
// pa4 = { => list3("curry", "chutney", "yak")}
println(pa4);
println:invoke(pa4); // [curry, chutney, yak]
```
Fast food!

```java
static Object fastAdd(int x, int y) {
    int z = x+y;
    if (((x ^ y) >= 0 && (x ^ z) < 0)) {
        println("oops, it's overflowing");
        return slowAdd(x, y);
    }
    return z;
}
```
Slowly brewed

```java
static Object slowAdd(Object x, Object y) {
    double xd = ((Number)x).doubleValue();
    double yd = ((Number)y).doubleValue();
    println("I'm hungry; is it done yet?");
    return xd + yd;
}
```
Moment of decision

```java
static boolean bothInts(Object x, Object y) {
    return x instanceof Integer && y instanceof Integer;
}
```
public static void main(String... av) {
    MethodHandle fastAdd =
        LOOKUP.findStatic(FastAndSlow.class, "fastAdd",
                         make(Object.class, int.class, int.class));
    MethodHandle slowAdd =
        LOOKUP.findStatic(FastAndSlow.class, "slowAdd",
                         make(Object.class, Object.class, Object.class));
    MethodHandle bothInts =
        LOOKUP.findStatic(FastAndSlow.class, "bothInts",
                          make(boolean.class, Object.class, Object.class));
    fastAdd = convertArguments(fastAdd, slowAdd.type());
    MethodHandle combo = guardWithTest(bothInts, fastAdd, slowAdd);
    println:invoke(combo, 2, 3));
    println:invoke(combo, 2.1, 3.1));
    println:invoke(combo, Integer.MAX_VALUE, -1));
    println:invoke(combo, Integer.MAX_VALUE, 1));
Demo sources...

NetBeans™ code demos are online here:

http://hg.openjdk.java.net/mlvm/mlvm/file/tip/netbeans/indy-demo

Outline of use:

hg clone http://hg.openjdk.java.net/mlvm/mlvm
cd mlvm/netbeans/indy-demo
vi nbproject/project.properties
ant run
Thank You

John Rose
John.Rose@sun.com

Christian Thalinger
Christian.Thalinger@sun.com

http://openjdk.java.net/projects/mlvm