Optimizing Dynamic Languages Using JSR292

Patrick Doyle
JIT compiler team
IBM Canada Laboratory
What is JSR292?

- Java Specification Request 292: 
  
  *Supporting Dynamically Typed Languages on the Java Platform*
Dynamically typed languages

- JVM is a popular platform to implement dynamic languages
  - There are whole conferences dedicated to this
  - Designed for JVM: Clojure, Groovy, Scala, …
  - Ported to the JVM: Python, Ruby, JavaScript, …

- JVM platform offers mature runtime support
  - Memory management
  - Class libraries
  - Dynamic compilation
  - Portability
Problem

- Looser / later type checking rules than Java
- Must forego unsuitable built-in features
  - … such as vtable-based virtual dispatches
  - … but we've spent 15 years optimizing those!
- Must work around some overly strict features
  - Linker and verifier do static type checking
- Must use custom idioms
  - Optimization is harder
  - Performance suffers
Outline

• Motivating example language
• Implementation #1: simple but slow
• Implementation #2: complex but fast
• Introduction to JSR292
• Implementation using JSR292: simpler and faster
Example language: CASPER

• \texttt{(CASCON Programming EnviRonment?)}
• Dynamically typed:

```python
def adder(x):
    if x is a String:
        return x.add(“CON”)
    else:
        return x.add(1)
```

```python
adder(2)      # returns 3
adder(“CAS”)  # returns “CASCON”
adder(stdout)  # throws NotUnderstood
```
JVM implementation

- We want to implement this on top of the JVM
- We want good performance
  - We're willing to compile CASPER to bytecode
  - … but in return, we expect Java-like performance!

```python
def adder(x):
    if x is a String:
        return x.add("CON")
    else:
        return x.add(1)
```
Casper-to-Java attempt #1

```python
def adder(x):
    if x is a String:
        return x.add("CON")
    else:
        return x.add(1)
```

```java
public static CasObject adder(CasObject x) {
    if (x instanceof CasString) {
        CasObject[] args = { x, CasRuntime.box("CON") };
        return x.lookup("add").send(args);
    } else {
        CasObject[] args = { x, CasRuntime.box(1) };
        return x.lookup("add").send(args);
    }
}
```

Runtime support code:
```java
public class CasObject {
    public class CasMessage {
        public static CasObject lookup(String name); }  
        public class CasMessage {
            public static CasObject send(CasObject[] args); }
        }
    }
```
Attempt #1 performance

- Call site must box arguments and pack into an array
- `send` is a virtual call to some nontrivial method
  - Might get devirtualized and inlined in a simple program
    - … but even this will fail for very polymorphic calls
    - … and it would still need to unpack / downcast / unbox
    - … and we can't pin all our hopes on the inliner
- This is a *lot* of gunk for the JIT to see through
  - … and the interpreter will be hopelessly slow
- **Nowhere near Java-like performance**
- Reflection would only make things worse
  - All of the above problems, plus more overhead
Attempt #1: bytecode for integer add

```
iconst_2
anewarray       CasObject
dup
iconst_0
aload_0
aastore
dup
iconst_1
iconst_1
invokestatic    CasRuntime.box(I)LCasObject;
aastore
astore_1
aload_0
ldc                 "add"
invokevirtual    CasObject.lookup(LString;)LCasMessage;
aload_1
invokevirtual    CasMessage.send([LCasObject;)LCasObject;
areturn
```

- Pack arguments into an array
- Box argument
What if we didn't need packing / boxing?

No packing!

No boxing!

Pass arguments as they are!
So what's the catch?

```java
aload_0
ldc "add"
invokevirtual CasObject.lookup(LString;)LCasMessage;
aload_0
iconst_1
invokevirtual CasMessage.send(LCasObject;I)LCasObject;
areturn
```

- **What is the signature for** `CasMessage.send`?
- **Answer: it must support every possible signature!**
- **How?**
  - Infinite amount of Java code
  - Dynamically generated bytecode
  - VM magic?
Dynamically generated bytecode

- Problem: signature differs for each call site
  - Example: integer add
    - add(LCasObject;I)LCasObject;
    - add(LCasInteger;LCasInteger;)LCasInteger;
    - add(I)I
    - ...
  - lookup must return *something* with a send method supporting the correct signature

- To avoid boxing, runtime must generate an *invoker class* per signature / callee pair
Bytecode using invokers

aload_0
ldc "add"
invokevirtual CasObject.lookup(LString;)LCasMessage;
checkcast Invoker1138
aload_0
iconst_1
invokevirtual Invoker1138.send(LCasObject;I)LCasObject;
return

- The good news: call sites look nice!
  - No boxing
  - No array packing
- What's the bad news?
Generating invokers

```java
class addInvoker42 extends Invoker1138 {
    public CasObject send(CasObject a, int b)
        { return CasInteger.add(((CasInteger)a).getInt(),b); } }

class addInvoker43 extends InvokerAA23 {
    public CasInteger send(CasInteger a, CasInteger b)
        { return CasInteger.add(a,b); } }

class addInvoker44 extends InvokerF00D {
    public int send(int a, int b)
        { return CasInteger.add(a,b); } }
```

- Need *lots* of these
  - \(O(n^2)\) in the size of the source code
- Huge pain compared with compiling Java
- This is what JSR292 renders unnecessary
What is JSR292?

- Positions JVM as target platform dynamic languages
- Core feature: the `MethodHandle` class
  - Conceptually: supports every possible signature
  - Practically: invokers are generated on demand
  - The perfect replacement for `CasMessage`!
Casper-to-Java with MethodHandle

def adder(x):
    if x is a String:
        return x.add("CON")
    else:
        return x.add(1)

public static CasObject adder(CasObject x) {
    if (x instanceof CasString) {
        return x.lookup("add").invoke(x, "CON");
    } else {
        return x.lookup("add").invoke(x, 1);
    }
}

Runtime support code:

```java
public class CasObject {
    public MethodHandle lookup(String name);
}
```

// no CasMessage--use MethodHandle instead
Bytecode using MethodHandle

aload_0
ldc "add"
invokevirtual CasObject.lookup(LString;)LMethodHandle;
aload_0
iconst_1
invokevirtual MethodHandle.invoke(LCasObject;I)LCasObject;
areturn

- Same call sequence as with invokers
- No other code to generate
  - The MethodHandles come from a reflection-like Java API
  - VM generates any invoker code itself internally
MethodHandle reduces overhead

- Almost all checking is done during MethodHandle creation
- Internally-generated invokers need no:
  - access checks
  - security checks
  - downcasting type checks
  - stack frames
  - class loading
  - verification
  - ...

MethodHandle reduces invokers

- Bytecoded invokers have static type annotations
  - Must satisfy Java linker and verifier rules
  - Some invokers differ *only* in their type annotations
    - eg. Invoker passing String can't be used to pass HashMap
      - identical bytecode
      - identical machine code
      - ... yet mismatched types will fail to link / verify!
- Internally-generated invokers can be shared aggressively
Benefits of JSR292

- MethodHandle is a powerful primitive
  - Overhead comparable to virtual call
  - Flexibility comparable to bytecoded invokers
    - … with a fraction of the generated code
    - … and it's all managed by the JVM
  - Much cheaper than reflection
    - No unnecessary unpacking / unboxing / downcasting
    - No access / security / type checks at invoke time

- Dynamic languages don't need custom idioms
  - Uniformity makes optimization easier