Dynamically Typed Languages on the Java Platform

Gilad Bracha
Computational Theologist
Sun Java Software

On Freedom of Choice

Any Customer Can Have Any Car
Painted Any Color That He Wants

On Freedom of Choice

Any Customer Can Have Any Car Painted Any Color That He Wants So Long As It Is Black

Henry Ford

On Freedom of Choice

The situation on mainstream computing platforms is similar

Any Customer Can Have Any Language She/He Wants So Long As It's C#

Anonymous

On Freedom of Choice

There is a even a perception that:

Any Customer Can Have Any Language She/He Wants So Long As It's Java

Malicious

Dynamic Languages on the JVM

This talk is about fixing that perception (and the underlying reality that gave rise to it)

Supporting Other Languages on The JVM

- What:
  - Some people program in several languages, especially scripting languages. We want to improve support for these on the JVM.

Supporting Other Languages on The JVM

- Why:
  - Useful for Java platform users for certain tasks
  - Broaden community

Supporting Other Languages on The JVM

- How:
  - ***Invokedynamic*** byte code

Supporting Other Languages on The JVM

- Today, OL implementations ride existing JVMs, for example:
  - Jython
  - Kawa
  - Groovy
  - ECMAscript
  - ....

Supporting Other Languages on The JVM

- Easy for single inheritance, single dispatch, statically typed OOPLs.
- Real interest is in languages that are different
  - Scripting languages are all dynamically typed
  - Most have multiple inheritance or mixins
- This can be challenging to do well

A Closer Look at Method Invocation

- JVM has 4 bytecodes for method invocation
  - ***invokevirtual***
  - ***invokeinterface***
  - ***invokestatic***
  - ***invokespecial***

Invokevirtual

General form is:

```
invokevirtual TargetObjectClass.methodDescriptor
```

where

```
MethodDescriptor -> methodName(ArgTypes) ReturnType
```

Invokevirtual

- Very close to Java programming language semantics
  - Only overloading (and generics) left to javac
  - Single inheritance, single dispatch, statically typed

Invokevirtual

- Very close to Java programming language semantics
  - Only overloading (and generics) left to javac
  - Single inheritance, single dispatch, ***statically typed***

Invokevirtual

- Very close to Java programming language semantics
  - Only overloading (and generics) left to javac
  - Single inheritance, single dispatch, ***statically typed***
  - Verifier will ensure that types are correct

And Here My Troubles Began

- Consider a trivial snippet of code in a dynamically typed language:

```
newSize(c)
// Collection has grown; figure out the next
// increment in size
{
    return  c.size() * c.growthFactor();
}
```

How to Compile this to the JVM?

In particular, how to compile the method invocations?

```
newSize(c)
// Collection has grown; figure out the next
// increment in size
{
    return  c.size() * c.growthFactor();
}
```

How to Compile this to the JVM?

```
newSize(c)
    // Collection has grown; figure out the next
    // increment in size
    {
        return  c.size() * c.growthFactor();
    }
    invokevirtual  IdontKnowWhatType.growthFactor() UnknownReturnType
```

How to Compile this to the JVM?

```
newSize(c)
   // Collection has grown; figure out the next
   // increment in size
   {
      return  ((Interface91)((Interface256) c).size() *
      ((Interface91)((Interface42) c).growthFactor();
   }
   invokeinteface  Interface42.growthFactor() Object
```

How to Compile this to the JVM?

- This is inefficient and brittle at best.
- Alternately, write your own interpreter and run it on top of the JVM.
- May Moore's law be with you.

Solution: *Invokedynamic*

- A loosely typed **invokevirtual**
- Target need not be statically known to implement method descriptor given in instruction
  - No need for a host of synthetic interfaces
- Actual arguments need not be statically known to match method descriptor

Invokedynamic

```
invokedynamic  Anytype.growthFactor() Object
```

Invokedynamic

- Actual arguments need not be statically known to match method descriptor
- What happens if they are wrong?

Invokedynamic

- Actual arguments need not be statically known to match method descriptor
- What happens if they are wrong? For example:

```
invokedynamic  LinkedList.get(int) Object
```

  When the argument is actually an object?

Invokedynamic

```
invokedynamic  LinkedList.get(int) Object
```

If the argument is actually an object, this might be used to convert a pointer to an integer. This undermines type safety an, most important, pointer/ memory safety.

Invokedynamic

```
invokedynamic  LinkedList.get(int) Object
```

If the argument is actually an object, this might be used to convert a pointer to a integer. This undermines type safety an, most important, pointer/memory safety.

Instead, cast at invocation time to ensure integrity.

No overhead for calling dynamically typed code.

Only a Partial Solution

- Calling Java platform libraries from scripting languages brings additional problems
- How do you resolve overloading?

Overloading

Given the code:

```
class Gourmand {
  Boolean eat(Food junk);
  Boolean eat(Fish freddie);
  Boolean eat(Mint thin);
}
```

How does one determine which method this dynamically typed code is calling:

```
var f = fetchFood();
(new Gourmand()).eat(f);
```

Overloading

One can resolve the method at run time, using the dynamic types of the arguments.

So, if `f` is an instance of `Salmon`, one chooses `eat(Fish freddie);`

Some would like the VM to do this for them, but this is too complex and brittle

Overloading

Instead, the call

```
eat(f)
```

is compiled as

```
invokedynamic Gourmand.eat(Object)Object
```

if no exact match is found, the invoke instruction traps to a user supplied handler

Overloading

The handler receives a reflective descriptor of the call, identifying the call site, the name and descriptor given at the call site, and an array of the actual arguments.

The handler can then choose to process the call as it wishes. In particular, it can choose to invoke a routine that resolves the overloading dynamically and caches the results based on the call site and arguments.

Only a Partial Solution

- No direct support for multiple inheritance or multiple dispatch
- General support is hard - each language has its own rules
- More complicated schemes possible (OMDB)

Invokedynamic

- In summary, ***invokedynamic***
  - is a natural general purpose primitive
    - Not tied to semantics of a specific programming language
    - Flexible building block for a variety of method invocation semantics
  - enables relatively simple and efficient method dispatch for dynamically typed languages

Conclusion

- Sun wants to see a variety of programming languages targeting the Java platform
  - Dynamically typed languages in particular; they fill a different niche than Java
- Improved support planned
  - Javascript and Groovy in the pipeline
  - ***invokedynamic*** planned (not yet approved)