Dynamically Typed Languages on the Java™ Platform

Gilad Bracha

Computational Theologist
Sun Microsystems

TS-3886

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 <u>So Long As It Is Black</u>"

– Henry Ford

We’d Like to Improve on That

- Programmers should be able to choose the right programming language for the right task

Support for Other Languages

- What:
  - Some people program in several languages, especially scripting languages; We want to improve support for these on the Java™ Virtual Machine (JVM™)
- Why:
  - Useful for Java™ platform users for certain tasks
  - Broaden community

Support for Other Languages

- Today, OL implementations ride existing Java VMs
  - Examples: Jython, Kawa
- 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 mix-ins
- This can be challenging to do well

Enter JSR 292

- Designed to:
    - Make it easier to implement scripting languages on the JVM software
    - Make such implementations much more efficient
- Two goals
    - Invokedynamic byte code
    - Improved Hotswapping

Agenda

Invokedynamic

Hotswapping

Agenda

Invokedynamic

Hotswapping

A Closer Look at Method Invocation

- JVM software has four bytecodes for method invocation
    - invokevirtual
    - invokeinterface
    - invokestatic
    - invokespecial

Invokevirtual

- General form is:
  - *invokevirtual TargetObjectType.methodDescriptor*
  - *MethodDescriptor → methodName(ArgTypes) ReturnType*
- Very close to Java programming language semantics
  - Only overloading (and generics) left to javac
  - Single inheritance, single dispatch, statically typed

Invokevirtual

- General form is:
  - *invokevirtual TargetObjectType.methodDescriptor*
  - *MethodDescriptor → methodName(ArgTypes) ReturnType*
- Very close to Java programming language semantics
  - Only overloading (and generics) left to javac
  - Single inheritance, single dispatch, statically typed

Invokevirtual

- General form is:
  - *invokevirtual TargetObjectType.methodDescriptor*
  - *MethodDescriptor → methodName(ArgTypes) ReturnType*
- 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 Java Virtual Machine?

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 Java Virtual Machine?

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();
    }
```

invokevirtual
    IdontKnowWhatType.growthFactor() UnknownReturnType

How to Compile This to the Java Virtual Machine?

Solutions are complex, involving many synthetic interfaces and casts.

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

invokeinterface
    Interface42.growthFactor() Object

How to Compile This to the Java Virtual Machine?

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
    - Instead, cast at invocation time to ensure integrity

invokedynamic Anyclass.growthFactor() Object

Invokedynamic

- Actual arguments need not be statically known to match method descriptor
  - Instead, cast at invocation time to ensure integrity
- Why?

Invokedynamic

- Actual arguments need not be statically known to match method descriptor
  - Instead, cast at invocation time to ensure integrity
- Why? Suppose argument types are wrong:

invokedynamic LinkedList.get(int) Object

When the argument is actually an Object

Invokedynamic

- Actual arguments need not be statically known to match method descriptor
  - Instead, cast at invocation time to ensure integrity
- Why? Suppose argument types are wrong:

invokedynamic LinkedList.get(int) Object

When the argument is actually an Object

- This could be used to convert a pointer to an integer
- Undermines type safety, and most important, pointer/memory safety

Invokedynamic

- Actual arguments need not be statically known to match method descriptor
  - Instead, cast at invocation time to ensure integrity
  - No overhead when calling dynamically typed code

Only a Partial Solution

- No direct support for multiple inheritance or multiple dispatch
  - General support is hard—each language has its own rules
- Calling Java platform libraries from scripting languages brings additional problems
  - How do you resolve overloading?
- However, invokedynamic is a useful primitive in most of these complex scenarios as well
- More complicated schemes possible (OMDB)

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 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

- Handler receives a reflective descriptor of the call, identifying:
  - Call site
  - Method name and descriptor at call site
  - Array of actual arguments
- Handler can process call as it wishes; in particular:
  - Can invoke routine that resolves overloading dynamically and caches results based on call site and arguments

Agenda

Invokedynamic

Hotswapping

Hotswapping

a.k.a. Reflective Program Change

- The ability to modify code on the fly
- Originates with Lisp, APL, Smalltalk
- Common feature in many scripting languages
- Very useful for:
    - Program development (e.g., fix-and-continue debugging)
    - Highly dynamic code that adapts to current conditions
- Addictive: Use it, and you’re hooked

Hotswapping

- Limited support in current Java VMs
  - Part of JVM Tool Interface (JVMTI)
- JVMTI may allow you to change the code in a method body
  - But, not always supported
- JVMTI will not support:
  - Changing method signatures
  - Adding/removing methods
  - Adding/removing fields
  - Changing class hierarchy

Hotswapping

- Not a feature of statically typed programming languages
  - Complex and costly to implement while maintaining type safety
  - Pay in time and/or space
    - Retypechecking codebase on every change is very time consuming
    - Incremental typechecking requires complex dependency management
      - Dependencies require space

Hotswapping

- So, what will JSR 292 do to change this?
- No firm commitment at this time!
- Most likely, allow hotswapping for dynamically typed languages
- Unlikely to allow hotswapping for statically typed languages

Hotswapping

- One approach:
  - Distinguish class files that are “Hotswappable”
  - Hotswappable classes can only be called via invokedynamic
    - No use of getfield, putfield, other invokes from outside the class
  - Unsafe use of a Hotswappable class fails dynamically in a controlled way
    - No core dumps or corruption of memory
  - Hence, no need for elaborate incremental typechecking/verification of clients of such classes

Hotswapping

- Too early to tell how this will play out
- Maybe we can do better, maybe we do worse
- The goal is to allow scripting languages to be implemented “natively” on the JVM software
- Simplify implementor's lives
  - Likely yield more good implementations for programmers to use
  - Potential for awesome performance over time

Summary

- Sun wants to see a variety of programming languages targeting the Java platform
  - Dynamically typed languages in particular; they fill a different niche the Java programming language
- Improved support planned
  - Javascript programming language and Groovy in the pipeline
  - JSR 292 starting up

For More Information

Useful links

- gilad.bracha@sun.com
- http://jcp.org/en/jsr/detail?id=292
- http://blogs.sun.com/gbracha/
- http://blogs.sun.com/roller/resources/gbracha/JAOO2005.pdf

Q&A

the POWER of JAVA™

Dynamically Typed Languages on the Java™ Platform

Gilad Bracha

Computational Theologist
Sun Microsystems

TS-3886