Structural Typing on the Java Virtual Machine with invokedynamic

by

Brian Diekelman

A thesis submitted in partial fulfillment for the degree of Bachelor of Science in the Department of Engineering And Computer Science

December 2012
This thesis describes the implementation of a structurally typed programming language and compiler for the Java Virtual Machine that uses the invokedynamic bytecode instruction and support library. The invokedynamic instruction itself is explained in detail along with descriptions of the call site bootstrapping and linking process. Details are provided on how to construct polymorphic inline caches inside of call sites targeting structural types using trees of method handles. The invokedynamic-based implementation is benchmarked against other structural typing techniques and outperforms Core Reflection API-based structural typing implementations by a factor of two.
Acknowledgements

I would like to thank Dr. T.K. Prasad, my thesis advisor.

Additionally, I would like to thank the members of my thesis committee, Dr. Michael Raymer and Dr. Prabhaker Mateti.
# Contents

Abstract i

Acknowledgements ii

List of Figures v

List of Listings vi

List of Tables vii

Abbreviations viii

1 Introduction 1

2 JVM Overview 3
  2.1 Class File Structure .................................................. 3
  2.1.1 Field and Method Descriptors .................................... 5
  2.2 Data Endpoints .......................................................... 5
    2.2.1 Operand Stack ...................................................... 5
    2.2.2 Local Variables ................................................... 6
    2.2.3 Fields ............................................................... 8
    2.2.4 Method Invocation ................................................ 9
  2.3 Bytecode Verification ................................................ 10
  2.4 Generalization ........................................................ 11

3 Invokedynamic 12
  3.1 Method Handles ......................................................... 12
  3.2 Method Handle Combinators .......................................... 14
    3.2.1 Filters ............................................................. 15
    3.2.2 Spreaders and Collectors ....................................... 17
    3.2.3 Guards ............................................................. 17
  3.3 Invocation Linking ..................................................... 17
    3.3.1 invokevirtual Invocation ........................................ 18
    3.3.2 invokedynamic Invocation ....................................... 18
    3.3.3 Call Site Relinking ............................................. 20
    3.3.4 Optimization ...................................................... 20

4 Structural Typing 21
  4.1 Nominal Typing ........................................................ 21
    4.1.1 Case Study: java.io.Closeable ................................ 22
  4.2 Structural Typing ..................................................... 23
    4.2.1 Structural Interfaces ............................................ 24
| Contents |
|-------------------------|----------|
| 4.3 Implementation       | 25       |
| 4.3.1 Compile-Time Slot Lookup | 25       |
| 4.3.2 Runtime Slot Lookup | 26       |
| 4.4 Inline Caching       | 28       |
| 4.4.1 Monomorphic        | 30       |
| 4.4.2 Polymorphic        | 30       |
| 4.4.3 Megamorphic        | 32       |
| 5 Kale Language          | 33       |
| 5.1 Overview             | 33       |
| 5.1.1 Types              | 33       |
| 5.1.2 Operators          | 34       |
| 5.2 Compilation          | 35       |
| 5.2.1 Invocations        | 36       |
| 5.3 Performance          | 40       |
| 6 Conclusion             | 42       |
| Bibliography             | 43       |
List of Figures

2.1 Data Endpoints ................................................................. 6
2.2 Invocation Bytecode ......................................................... 9

3.1 Method Handle Composition .............................................. 16
3.2 invokevirtual Linking ..................................................... 18
3.3 invokedynamic Linking .................................................... 19

4.1 Single Inheritance Memory Layout .................................... 26
4.2 Structural Inheritance Layout .......................................... 27
4.3 Method Handle PIC ......................................................... 31
## List of Tables

2.1 Type Descriptors ........................................... 4
2.2 Method Descriptors ......................................... 5
2.3 Field Instruction Descriptors .............................. 11
3.1 Method Handle Descriptors ................................. 14
3.2 `getfield` Descriptors ...................................... 16
5.1 Invocation Benchmarks ...................................... 41
Abbreviations

JVM    Java Virtual Machine
JDK    Java Development Kit
JSR    Java Specification Request
MLVM   Multi Language Virtual Machine
API    Application Programming Interface
LIFO   Last In First Out
Chapter 1

Introduction

The Java Virtual Machine (JVM) has become ubiquitous in enterprise applications for its platform portability, performance, and stability. Companies such as Google [1], Facebook [2], and Twitter [3] all deploy the JVM at scale in their application stack. The benefits of the JVM have enticed developers to build implementations of Python, Ruby, and over 100 domain specific languages that run on the JVM [4]. Languages like Jython and JRuby that target the JVM get access to generational garbage collection, native threading, and a highly optimized Just In Time (JIT) compiler.

Regardless of the current level of adoption of other languages, the JVM was designed primarily to run Java. All the concepts of Java – classes, exceptions, strong typing, and the distinction between primitive and reference types – are all represented in code targeting the JVM. Therefore, language implementers are forced to express the concepts of their language in terms of JVM structures. This becomes problematic when the language being implemented doesn’t conform to JVM function invocation and typing semantics. In order for languages using non-Java semantics to run on the JVM, significant workarounds must be employed. These workarounds typically increase implementation complexity, cause code bloat, and decrease runtime performance.

However, JVM 7 includes a new feature, invokedynamic, that makes the JVM a more flexible and adaptable language host. invokedynamic is a new bytecode instruction and runtime support library that together allow a lower level of access to class loading and linking operations within
the JVM. Language features like structural typing, which defines type equivalence based on types defining the same set of methods, can use this new link access to attain a level of performance that was not previously possible.

Details of the current linking and invocation semantics of the JVM are described in Chapter 2. `invokedynamic` itself is described in detail in Chapter 3, and its integration into a structural type system is described in Chapter 4. Finally, a compiler for a language that supports structural typing for the JVM through the use of `invokedynamic` is described in Chapter 5.
Chapter 2

JVM Overview

The Java Virtual Machine (JVM) is a stack-based, hardware agnostic virtual machine that is defined by the Java Virtual Machine Specification [5]. JVM executable code is defined in terms of portable bytecode [5, 4.10] stored in the class file format.

2.1 Class File Structure

The class file format was originally designed to be a binary storage format for compiled Java code. Due to this, there is a one to one correspondence between structure in a Java source file and corresponding elements in a class file. Each class file defines one class, which must extend exactly one other class (the superclass), and can implement zero or more interfaces. Each class can contain zero or more fields and zero or more methods. Given an input Java source file such as Listing 2.1, a Java compiler would generate a class file output with bytecode representing method add similar to Listing 2.2.

```
package test;

class SomeClass {
    int value;
    int add(int b) {
        return value + b;
    }
}
```

Listing 2.1: Java before compilation
Note that even though no *extends* declaration was used when declaring the class `SomeClass`, since every class must extend another class, the compiler automatically generates bytecode that extends the base class in the Java object system, `java.lang.Object`. All classes, fields, and methods within the JVM are strongly typed with symbolic type descriptors [5, 4.3]. There are two classifications of types within the JVM: *primitive* and *reference* types. Primitive types are unboxed numeric types such as `int` and `boolean` [5, 2.3]. Reference types refer to dynamically created instances of a class. Table 2.1 shows the format of the type descriptors that will be used throughout this document.

<table>
<thead>
<tr>
<th>Type</th>
<th>Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>I</td>
</tr>
<tr>
<td>boolean</td>
<td>Z</td>
</tr>
<tr>
<td>void</td>
<td>V</td>
</tr>
<tr>
<td>classref</td>
<td>Lclassname;</td>
</tr>
</tbody>
</table>

Table 2.1: Type descriptors for common JVM types [5, 4.3.3].
2.1.1 Field and Method Descriptors

A field descriptor defines the name and type of a class field. As shown in line 3 of listing 2.2, the value field of the class is defined with the type descriptor I, which Table 2.1 defines as int (a 32-bit two’s-complement integer).

A method descriptor is composed of the type descriptors of its parameters and return type [5, 4.3.3]. As shown in line 4 of Listing 2.2, the method add(I)I takes an int parameter and returns an int. Example mappings between Java method declarations and the resulting method descriptors are provided in Table 2.2.

<table>
<thead>
<tr>
<th>Java Declaration</th>
<th>Method Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>void (int a, int b)</td>
<td>(II)V</td>
</tr>
<tr>
<td>boolean (int a, String b)</td>
<td>(IIjava/lang/String;)Z</td>
</tr>
</tbody>
</table>

Table 2.2: Method Descriptors.

2.2 Data Endpoints

In place of directly addressable memory, all data transfers within the JVM occur between so-called data endpoints [6]. Data endpoints are strongly typed memory stores that include: an operand stack, a local variable array, class fields, and class methods. The flow of data between data endpoints is represented in Figure 2.1.

2.2.1 Operand Stack

The operand stack is a fixed-sized, Last In First Out (LIFO) data structure [5, 2.6.2] used to perform computation and store intermediary values between other data endpoints. Each method
has access to its own private operand stack during execution, the size of which is determined
by the maxStack class file attribute that is specified at compile time.

Values are pushed onto the stack, either explicitly through load instructions or as a result of
other instruction execution such as a return value being loaded to the stack after a method
invocation. Once values are on the stack, arithmetic can be performed using instructions like
iadd and isub (add and subtract, respectively). Listing 2.3 shows example bytecode that loads
values onto the stack and adds them.

```
1 ldc 2  // load constant 2 onto the stack
2 ldc 3  // load constant 2 onto the stack
3 iadd  // add the top two values on the stack
```

Listing 2.3: Operand stack arithmetic

### 2.2.2 Local Variables

During execution, a method has access to a fixed number of local variable slots [5, 2.6.1]. Listing
2.4 shows an example Java method that annotates how the locals array is populated.

```
static int add( int b ) {  // locals [0] = b
    int i = 7;  // locals [1] = i
    int j = 6;  // locals [2] = j
    return b + i + j;
}
```
Local variables are transferred to and from the operand stack with the `load` and `store` opcodes.

Figure 2.4 shows how the locals array could be allocated given local variable declarations. Listing 2.5 details how the locals array would interact with the operand stack as depicted in Listing 2.4.

```
    ldc 7    // load the constant 7 onto the stack
    istore 1 // store 7 to locals[1]
    ldc 6    // load the constant 6 onto the stack
    istore 2 // store 6 to locals[2]
    iload 0  // load b from locals[0]
    iload 1  // load i from locals[1]
    iload 2  // load j from locals[2]
    iadd     // add i + j
    iadd     // add b to result of i + j
    iret     // return result
```

**Listing 2.5: Load and store**

Each load instruction in Listing 2.5 is prefixed by an 'i' to indicate the type of value being loaded. Integer values require an i prefix, while references require an a prefix.

The size of the locals array is statically defined within bytecode by the `maxLocals` attribute of the method. Attempting to access an index outside of this limit at runtime causes an exception to be thrown.

\[1\] Note that compiler optimizations would remove the redundant loads and stores in Listing 2.5, but it is instructive to retain them.
2.2.3 Fields

Field values of class instances are loaded onto the stack with the `getfield` instruction and written with the `putfield` instruction. Listing 2.6 defines a class `A` with an integer field `value` and Listing 2.7 shows the bytecode necessary to read from the `value` field.

```java
package test;

class A {
    int value;
}
```

**Listing 2.6: Java class fields**

```java
aload [objectref:A]
ggetField test/A.value I
```

**Listing 2.7: Load the value of a field onto the stack**

After the bytecode in Listing 2.7 executes, the value of the `value` field of class `A` will be on the top of the stack. The bytecode in Listing 2.8 consumes the two values loaded on the stack and sets the value of the `value` field of class `A` to 7.

```java
aload [objectref:A]
ldc 7
putfield test/A.value I
```

**Listing 2.8: Set the value of a field**

Access to static fields is performed with the `getstatic` and `putstatic` instructions, which behave identically to `getfield` and `putfield` except they do not require an object reference to be loaded onto the stack.
CHAPTER 2. JVM OVERVIEW

2.2.4 Method Invocation

Method invocation is performed by pushing arguments onto the operand stack, then using one of the invocation instructions. Invocation instructions pop all arguments off the operand stack, including the object reference if present, and push the return value if one exists for the target method. Each invocation instruction is referred to as a call site, and the object reference the method is being called on is referred to as the receiver. Listing 2.9 shows different forms of declared methods in Java source, and Figure 2.2 shows Java invocations of those methods and the corresponding invocation bytecode.

```
package test;

class A {
    static int a() {
        ...
    }
    int b() {
        ...
    }
    int c(int i) {
        ...
    }
}
```

Listing 2.9: Java class methods

```
a();
ob().b();
ob.c(9);
```

```
invokestatic test/A.a ()I
aload [obj]
invokevirtual test/A.b ()I
aload [obj]
ldc 9
invokevirtual test/A.c (I)I
```

Figure 2.2: Mapping Java source to bytecode

The simplest form of invocation is a static method invocation using the invokestatic instruction, which requires just arguments and a symbolic link to the target method as shown on line 1 of Figure 2.2. Virtual methods (methods declared without the static keyword), are called with the invokevirtual instruction, which requires the object reference of the receiver be pushed
on the stack along with the arguments. If the receiver is an interface, the `invokeinterface` instruction must be used as shown in Listings 2.10 and 2.11.

```java
package example;
interface Testable {
    void test(String s);
}
class SomeClass implements Testable {
    void test(String s) {
    }
}
```

Listing 2.10: Java interface declaration

```java
invokeinterface example/Testable test (Ljava/lang/String;)V
```

Listing 2.11: Java interface invocation

If `invokeinterface` is used with an object reference that does not implement the specified interface, a `ClassVerifyException` is thrown by the JVM at runtime.

More details about the invocation bytecode instructions can be found in Bill Venner’s 1997 JavaWorld article [7].

### 2.3 Bytecode Verification

All data endpoint access instructions embed not only the symbolic link to their target, but also the target’s type descriptor. Additionally, instructions generally require that a specific combination of operands with specific types are loaded onto the stack in order for them to execute without error. After the class is loaded into memory and parsed, and before it is made available for execution to other classes within the JVM, the class is run through a verification process to statically prove each instruction has been provided its required operands in the correct order. Any deviations from the required contract throw a `ClassVerifyException` at runtime.
2.4 Generalization

All data endpoint access discussed in this chapter goes through the same process: verify permissions, verify target type, verify operand types, and lookup receiver. The field access instructions can be modeled as function, as shown in Table 2.3, where $T$ is the field type and $R$ is the receiver type.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>getfield</td>
<td>$(R)T$</td>
</tr>
<tr>
<td>putfield</td>
<td>$(R,T)V$</td>
</tr>
<tr>
<td>getstatic</td>
<td>$(T)$</td>
</tr>
<tr>
<td>putstatic</td>
<td>$(T)V$</td>
</tr>
</tbody>
</table>

Table 2.3: Field instructions modeled as functions.

With all data endpoint JVM instructions generalized to function invocations, instructions can be composed at runtime, as Chapter 3 will discuss in detail.
Chapter 3

Invokedynamic

Though non-Java languages have been running on the JVM since its initial public release, no real effort was made by the engineers within Sun (now Oracle) to make the JVM a hospitable host to languages with semantics that differed from Java. As more production languages (such as JRuby) used JVM-based runtimes, it became apparent to Sun that more work needed to be done at the JVM level to support these languages. The goal was to make the JVM more flexible and adaptable while still retaining the strong type verification required to enforce Java’s type system. In order to accomplish this, fundamental changes to the JVM were needed. These modifications to core JVM functionality, a so-called “renaissance” of the JVM, were created as a branch of the OpenJDK project named the Da Vinci Machine Project in mid-2006 and formalized under JSR-292 [8]. The finalized implementation was included in the release of JDK 7. The modifications included low cost function pointers in the form of method handles, new method linking options with the invokedynamic instruction, and changes to the runtime optimizer.

3.1 Method Handles

Before JSR-292, the smallest composable unit within the JVM was the class. The only available facility to pass invokable references to methods was the Core Reflection API, which was not designed for performance intensive applications. The Core Reflection API was designed as a
meta layer to provide introspection to classes and their declared fields and methods at runtime.

The runtime support for Reflection existed outside of the reach of the JVM optimizer, which meant that reflective invocations could not benefit from optimization heuristics such as inlining [9]. Reflective invocations performed access control checks at each invocation, which resulted in significant overhead at runtime [10]. Instead of attempting to redefine the behavior of the Core Reflection API and break backwards compatibility, the JSR-292 Expert Group elected to create a new abstraction. Instead of separate classes to model fields and methods like the existing Core Reflection API, a single interface was created to abstract both field modification and method invocation bytecode instructions: the method handle.

Method handles are function pointers represented as a method descriptor that encapsulate field modification and method invocation operations. They are represented in the Java standard library in JDK 7 by the MethodHandle class in the java.lang.invoke package. Unlike the Core Reflection Field and Method classes, a method handle does not exist to provide metadata or perform runtime introspection. A method handle is meant purely as an efficient means of invoking a functional representation of a bytecode instruction. Listing 3.1 defines a class, A with a single field and three methods that are resolved using a MethodHandles.Lookup in Listing 3.2.

```java
package test;

class A {
    int value;
    static int a() { ... }
    int b() { ... }
    int c(boolean x) { ... }
}
```

Listing 3.1: Java field and method lookup targets
MethodHandles.Lookup lookup = MethodHandles.lookup();

MethodHandle mh1 = lookup.findGetter(A.class, "value", methodType(int.class));

MethodHandle mh2 = lookup.findStatic(A.class, "a", methodType(int.class));

MethodHandle mh3 = lookup.findVirtual(A.class, "b", methodType(int.class));

MethodHandle mh4 = lookup.findVirtual(A.class, "c", methodType(int.class, boolean.class));

Listing 3.2: Reflective lookup using MethodHandles.Lookup

The descriptors of the method handles acquired in Listing 3.2 are listed in Table 3.1.

<table>
<thead>
<tr>
<th>Method Handle</th>
<th>Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>mh1</td>
<td>(Ltest/A;)I</td>
</tr>
<tr>
<td>mh2</td>
<td>()I</td>
</tr>
<tr>
<td>mh3</td>
<td>(Ltest/A;)I</td>
</tr>
<tr>
<td>mh4</td>
<td>(Ltest/A;Z)I</td>
</tr>
</tbody>
</table>

Table 3.1: The descriptors of the method handles acquired in Listing 3.2.

Each MethodHandle can then be called using its invoke(Object...) method, which performs the underlying field access or method invocation.

### 3.2 Method Handle Combinators

Now that the previously disjoint operations are unified with a common functional abstraction, they can be combined into composite operations. The potential of method handles is to generate and reconfigure code at runtime without generating and loading raw bytecode into a class loader. Instead, method handles can be composed with one another — a function pointer that calls another function pointer — and integrated with existing class methods to allow flexible code
structures constructed solely from method handles that the native code optimizer within the JVM is aware of and able to optimize.

Every bytecode instruction involving data endpoints is representable as a method handle. These baseline method handles are provided by the `MethodHandles` class included in the `java.lang.invoke` package in JDK 7. The `MethodHandles` class provides method handles that filter arguments and return values, spread and collect vararg arguments, and conditionally branch between method handles.

### 3.2.1 Filters

The `filterArguments()` and `filterReturnValue()` in the `MethodHandles` class enable method handles to be adapted to call site descriptors or linked together. For example, given the classes `A`, `B`, and `C` in Listing 3.3, evaluating the expression `a.b.c.value` would require the bytecode listed in Listing 3.4.

```java
class A {
    B b;
}
class B {
    C c;
}
class C {
    int value;
}
```

Listing 3.3: Classes to execute `getfield` instructions against

```java
aload [a]
gtextfield A.b
gtextfield B.c
gtextfield C.value
```
This behavior can now be encapsulated into a single method handle by composing multiple method handles and filtering their return values. Table 3.2 lists equations that describe each method handle performing getfield operations for \texttt{a.b.c.value}.

<table>
<thead>
<tr>
<th>Bytecode Operation</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>getfield A.b</td>
<td>( m^d_0 = (A) \rightarrow B )</td>
</tr>
<tr>
<td>getfield B.c</td>
<td>( m^d_1 = (B) \rightarrow C )</td>
</tr>
<tr>
<td>getfield C.value</td>
<td>( m^d_2 = (C) \rightarrow I )</td>
</tr>
</tbody>
</table>

Table 3.2: Equation representation of the method handle descriptors.

The return value of each method handle is then filtered through the next method handle in the chain:

\[
m^d_0(A) \rightarrow m^d_1(B) \quad m^d_1(B) \rightarrow m^d_2(C) \quad m^d_2(C) \rightarrow I
\]

After substitution, the method handle tree simplifies down to a method handle that takes an argument of type \( A \) and returns the \texttt{int} value of the \texttt{value} field in class \( C \): \( m^d_0(A) \rightarrow I \).

A visual representation of this filtering process is depicted in Figure 3.1.

![Figure 3.1: A tree of method handles simplify through substitution.](image)

Filtering allows one complex operation to be wrapped into one method handle that can then be invoked, passed as a single object reference to other code, or composed with other method handles for even more complex behavior.
3.2.2 Spreaders and Collectors

A method handle with the descriptor \((A, B, C)D\) could be adapted to a more general method handle with the descriptor \((Object[]) Object\) using the `asVarArgsCollector(Object[].class)` method of the `MethodHandle` class. This collects the arguments \(A\), \(B\), and \(C\) into the `Object[]` argument of the adapted method handle. The inverse operation can be performed with the `asSpreader(Class<?>, int)` method of the `MethodHandle` class, which takes all arguments from the provided integer offset and "spreads" them – distributes them as individual parameters – back to the original \((A, B, C)D\) form.

3.2.3 Guards

Guards enable the composition of a traditional if-then-else statement using method handles. Constructing a guard is done by the `guardWithTest` method within the `MethodHandles` class which requires three method handles as arguments. First, a guard predicate which returns a `boolean` type. Second, a target method handle to invoke if the guard predicate is true. Third, a fallback method handle to invoke if the guard predicate is false. Using another guard as the fallback method handle enables the construction of a tree of conditional branches.

3.3 Invocation Linking

The inflexibility of the JVM prior to JDK 7 stems from the rigidity of the invocation instructions. The modifications implemented in the Da Vinci Machine Project that make this linkage process more flexible come in the form of a new invocation instruction, `invokedynamic`. 
3.3.1 invokevirtual Invocation

As detailed in Chapter 2, the JVM uses an internal symbolic linking process to link an invocation instruction, a call site, to the target class method. During this process the type descriptor embedded in the call site is compared to the type descriptor of the target method and an exception is thrown by the JVM if the type descriptors are not compatible. A depiction of the linking process for the invocation instruction in Listing 3.5 is shown in Figure 3.2.

```java
invokevirtual path/to/OtherClass.otherMethod (I)V
```

Listing 3.5: virtual method invocation bytecode

![Figure 3.2: JVM runtime internally resolves symbolic link.](image)

3.3.2 invokedynamic Invocation

Instead of a hardwired symbolic link to the target method, invokedynamic call sites embed a symbolic link to a static bootstrap method which returns a CallSite object. The CallSite object contains a MethodHandle which points to the invocation target method. Simply put, instead of specifying a target method, an invokedynamic call site specifies "who to ask" to get a target method. Along with the bootstrap method symbolic link, invokedynamic call sites also embed the target method name and type descriptor, which acts as a contract that must be fulfilled by the method handle in the call site returned by the bootstrap method. An example
invokedynamic instruction and corresponding linkage visualization are depicted in Listing 3.6 and Figure 3.3, respectively.

```java
void someMethod(OtherClass o) { 
    invokedynamic otherMethod(OtherClass, I)V 
                path/to/LanguageRuntime,  
                "bootstrap",  
                (Lookup, String, MethodType) CallSite 
}
```

**Figure 3.3:** JVM runtime defers to user code bootstrap to resolve symbolic link.

```java
Listing 3.6: dynamic invocation bytecode
```

The bootstrap method uses the reflective `MethodHandle.Lookup` class to obtain method handles, and can optionally perform any combination of method handle adaptation as discussed in Section 3.2.1. The bootstrap method is defined in JVM bytecode just like any other class, as opposed to the internal JVM process that links the other invocation instructions. Listing 3.7 defines an example bootstrap method.

```java
class LanguageRuntime {
    static CallSite bootstrap( Lookup lookup,  
            String methodName,  
            MethodType type ) {  
        MethodHandle mh = // resolve a MethodHandle ...  
        return new MutableCallSite(mh);
```
3.3.3 Call Site Relinking

CallSite is an abstract base class extended by ConstantCallSite and MutableCallSite. A ConstantCallSite cannot be re-targeted to another method handle, while the MutableCallSite can be updated with its setTarget(MethodHandle) method. A ConstantCallSite generally has better performance at the price of flexibility, but the MutableCallSite is useful in building adaptive call sites that are able to rebuild themselves in response to the receivers invoked on them.

3.3.4 Optimization

The primary feature that sets method handles and the invokedynamic infrastructure apart from other JVM features like the Core Reflection API is that method handles are designed to be able to take advantage of many of the optimizations applied to normal bytecode [11]. Specifically, method handles can be inlined and constant folded as the method handle tree symbols are evaluated and optimized at runtime.
Chapter 4

Structural Typing

A type system that requires equivalent types to have the same declared name, regardless of whether they define identical fields and methods, is referred to as a **nominal (or nominative)** type system. A type system that ignores the declared name of types and only requires equivalent types to share common field or method declarations is referred to as a **structural** type system.

4.1 Nominal Typing

As described in Chapter 2, the JVM uses nominal types throughout. Type names are declared in Java source, embedded into bytecode instructions, and enforced at runtime by the JVM. As shown in Listing 4.1, even though classes A and B are structurally identical, they are not considered equivalent within Java or by the JVM at runtime, and therefore cannot be used interchangeably.

```java
class A {
    void test() { ... }
}
class B {
    void test() { ... }
}
void m1(A a) {
    a.test();
    m2(a); // illegal
}
void m2(B b) {
```
4.1.1 Case Study: java.io.Closeable

Prior to the release of J2SE 1.5 in 2004, the Java standard library developers noticed they had several classes in the java.io package that all had close() methods. They wanted to be able to write generalized code that could operate on all of these classes, but the classes all had different hierarchies – they extended different base classes – so they could not be addressed as one common type. To solve this problem, the java.io.Closeable interface was introduced. This interface defines only a close() method that throws an IOException, and is meant to enable objects that represent resources such as files and network connections to be closed without the caller knowing anything about the class implementation other than the fact that it implements close().

However, this presented a problem when dealing with the large collections of code already written. Existing classes that had close() methods but did not explicitly implement the Closeable interface – because it didn’t exist when the code was written – could not be passed as an argument to code expecting an instance of Closeable. Due to Java’s nominal type system, existing code would have to be modified to add the implements Closeable declaration to every relevant class definition and then be recompiled and released as a new version.

One solution to the Closeable problem would be to create a way for classes that do not explicitly implement the Closeable interface to still be treated as equivalent to the Closeable interface.
as long as they implement that single `close()` method. That concept is a form of structural typing.

## 4.2 Structural Typing

Listing 4.2 demonstrates both the benefits and potential pitfalls of structural typing. Similar to Listing 4.1, two structurally equivalent types are defined. The difference is that, since `Cowboy` and `Rectangle` are structurally identical and they are defined within a structural type system, they can be used interchangeably on lines 9 and 13.

```java
class Cowboy {
    void draw() { ... }
}
class Rectangle {
    void draw() { ... }
}
void drawGun(Cowboy cowboy) {
    cowboy.draw();
    paint(cowboy);  // legal
}
void paint(Rectangle rect) {
    rect.draw();
    drawGun(rect);  // legal
}
```

Listing 4.2: Structural types

This approach solves the `java.io.Closeable` problem. Code could be written within this system that operated on both the `Cowboy` and `Rectangle` types uniformly, even though they have no explicitly declared relation to one another. New interfaces can be written that existing classes implicitly implement without modification. However, this interchangeable usage should immediately be cause for concern. The `draw()` method in Listing 4.2, while lexically identical in
each class, has a different meaning within the domain of each class. In the context of a Cowboy, to `draw()` means to draw a gun. In the context of a Rectangle, `draw()` means to render itself. This is an example of the ambiguity of the English language leaking into what should be a more formal definition within a programming language. This *accidental equivalence*, an edge case within structural type systems, can be mitigated through a hybrid use of structural typing with *structural interfaces*.

### 4.2.1 Structural Interfaces

There is a design compromise between structural typing and nominal typing that solves the `java.io.Closeable` problem while minimizing accidental equivalence. Described with Java-like syntax in Listing 4.3, the hybrid approach uses nominal typing to determine equivalence of two classes and structural typing to determine whether a type conforms to an interface.

```java
class Cowboy {
    void draw() { ... }
    int getHeight() { ... }
}
class Rectangle {
    void draw() { ... }
    int getHeight() { ... }
}
interface Height {
    int getHeight();
}

Listing 4.3: Structural interface
```

In Listing 4.3, Cowboy and Rectangle both conform to the Height interface by implementing the `getHeight()` method and can be provided to any receiver that is expecting an instance of Height. However, even though Cowboy and Rectangle have identical structures they are
not equivalent to each other and cannot be used interchangeably, thereby limiting accidental equivalence errors.

4.3 Implementation

The implementation challenges of structural interfaces stem from the fact that each concrete type (a type that corresponds to a specific memory layout) has the potential to implicitly implement a large number of interfaces. An interface like `java.io.Closeable` in a structural type system could be implemented by hundreds of concrete types, so when a function accepts an instance of `java.io.Closeable` as a parameter and calls its `close()` method the compiler has no idea which of the hundreds of implementing types will be passed to the function at runtime. If the fields and methods of a concrete type are generalized as slots (an offset within the memory layout of the concrete type), the problem can be described as whether the compiler can calculate the offset of a slot based only on information available at the call site at compile time, or if the offset of the slot cannot be determined until runtime.

4.3.1 Compile-Time Slot Lookup

Each name of a class in Java corresponds to one and only one memory layout of the fields and methods of that class. Listing 4.4 defines several Java classes, and Figure 4.1 shows an approximation of the corresponding memory layout.

```java
class Base {
    void sayHello () { ... }
}
class Class1 extends Base {
    void m1() { ... }
}
```
Given a pointer to an instance of `Base`, the compiler knows at compile time that `sayHello()` will always be located at offset [0]. The same relationship exists between `Class2` and its subclass, `Class3`. Given a pointer to any class that extends `Class2`, `m2()` will always be at offset [1]. Each subclass aligns its lower offsets with its base class structure, then uses its higher offsets for its own structure. This predictability enables a compiler to calculate the offset of any field or method of a class and generate a simple indirect load or jump instruction to access it.

### 4.3.2 Runtime Slot Lookup

Listing 4.5 shows pseudocode of two interfaces and two classes. `Class1` and `Class2` both implement the `Stream` interface by implementing `close()`. `Class2` also implements the `Gettable`
interface by implementing `get()`.

```java
interface Gettable {
    void get();
}

interface Stream {
    void close();
}

class Class1 {
    void close() { ... }
}

class Class2 {
    void close() { ... }
    void get() { ... }
}

void call(Stream stream) {
    stream.close();
}
```

**Listing 4.5:** Pseudocode: Java with structural types

<table>
<thead>
<tr>
<th>Offset</th>
<th>Gettable</th>
<th>Stream</th>
<th>Class1</th>
<th>Class2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>get()</td>
<td>close()</td>
<td>close()</td>
<td>close()</td>
</tr>
<tr>
<td>[1]</td>
<td></td>
<td></td>
<td></td>
<td>get()</td>
</tr>
</tbody>
</table>

**Figure 4.2:** An abstract representation of class memory layout.

When calling into concrete types, the offset of a given method was known at compile time. With structural typing this is no longer the case, as evident by the memory layout of `Class2` in Figure 4.2. Since `Class2` implements both the `Gettable` and `Stream` interfaces it must allocate a slot in its memory layout for both the `get()` and `close()` methods. However, both of those methods are defined at offset [0] in their respective interfaces. If a compiler places the `close()` method at offset [0] of `Class2`, any call sites receiving an instance of `Class2` as a `Gettable` cannot predict at compile time the offset at which the `get()` method will be located. The
inverse is also true if the `get()` method was placed at offset [0]. This problem is aggravated when more than two interfaces are considered across multiple concrete types.

The offset that must be resolved on the receiver from the call site on line 15 cannot be determined through static analysis. The call site could be provided by either `Class1` or `Class2` at runtime. Regardless of which type is provided, the call site must be able to efficiently lookup the offset for the `close()` method on the provided type and invoke it. This ad-hoc type introspection requirement has the potential to introduce significant overhead per invocation.

Prior to `invokedynamic`, the rigid nature of the invocation bytecode instructions meant the options to lookup and call methods on types at runtime were limited. Implementing a feature like structural typing on the JVM required either reflection or a code generation technique that generated inline type checks at the call site at compile time [12]. Neither option was preferable, since reflection introduced significant overhead and the code generation technique cannot account for types that are unknown at compile time.

With `invokedynamic` method lookup overhead can be reduced by having the call site itself cache references to methods by receiver type as they are invoked. This approach has the potential to completely eliminate the method lookup as long as the receiver type is in the call site cache.

### 4.4 Inline Caching

An *inline cache* is a dispatch table embedded in a call site that builds an associative cache of receiver type metadata in order to optimize future invocations. Inline caching was first developed for use in Smalltalk implementations in the 1980s [13] in order to improve dynamic
invocation performance. Listing 4.6 defines an interface call site that could benefit from inline caching.

```java
interface Gettable {
    void get();
}
class A {
    void get() { ... }
}
class B {
    void get() { ... }
}
class C {
    void get() { ... }
}
void call( Gettable g ) {
    g.get();
}
```

Listing 4.6: An interface call site

On the first invocation of `call(Gettable)` in Listing 4.6, the inline cache of call site `g.get()` will be empty. If, for example, an instance of class `A` was passed as the parameter on the first invocation, a full lookup would have to be performed to examine the methods implemented in class `A` and find `get(Gettable)`. Once it is resolved, however, the offset of `get()` within class `A` will be added to the inline cache as an association between the receiver type `A` and the method handle of `get()` within type `A`.

Depending on how many receivers a call site is associated with, it can be referred to as either

*monomorphic, polymorphic,* or *megamorphic.*
4.4.1 Monomorphic

According to research by Oracle’s HotSpot engineering team, 90% of call sites only target one concrete method over their entire lifetime [14]. These call sites, called monomorphic, retain a reference to the previously invoked receiver type. Upon subsequent invocation, the previous receiver type is compared to the new receiver type and, if they match, the cached receiver metadata can be used to jump directly to the target method.

The design decisions inherent to monomorphic call sites all revolve around the caching heuristics. Primarily, what happens on a cache miss? If a call site replaces its cached receiver type each time the new receiver doesn’t match, the overhead of cache maintenance could dominate execution time if a call site is being called with two types in an alternating invocation pattern. Conversely, choosing to not replace the cached receiver type could miss an optimization opportunity. One approach to avoid these issues, developed as an extension to the monomorphic inline cache by Sun Microsystems for the Self project, is to simply grow the cache to accommodate new receiver types [15].

4.4.2 Polymorphic

A polymorphic inline cache is an inline cache that stores multiple receiver types. Pseudocode for a polymorphic inline cache for the call site in Listing 4.6 is demonstrated in Algorithm 1.
switch `receiver.type` do
  case A A.get();
  case B B.get();
  case C C.get();
  otherwise lookup(receiver);
endsw

**Algorithm 1:** Inline cache method dispatch

For the roughly 10% of call sites that are invoked against more than one receiver type, a polymorphic inline cache is allocated with initial space to store several receiver types. If a new receiver type is used at the call site, it falls through to the slower `lookup()` function which performs a linear search on the receiver type for the target method, then adds it to the cache.

This behavior can be implemented with a tree of method handles as depicted in Figure 4.3.

![Figure 4.3: A polymorphic inline cache implemented with nested method handles.](image-url)
Constructed with the `guardWithTest()` factory method in the `MethodHandles` class, the guard is bound to the `isInstanceOf(Class)` method of the `java.lang.Class` metaclass of the receiver. If the guard succeeds, it immediately invokes a method handle that references the concrete method. If the guard fails, the fallback of the guard is the guard of the next receiver type check. There are several optimization variations that influence how this tree can be constructed. New types can be added to the root of the tree or inserted before the final `lookup(receiver)` call. Invocation counts could be tracked, bubbling up the most invoked receiver type to the root of the tree so it is checked first in order to speed up dispatch time. However, obviously the tree cannot grow without bound. The inline cache must be designed to handle the 1% of cases where an unmanageable number of types flow through one call site.

### 4.4.3 Megamorphic

A call site that handles an inordinate amount of receiver types is referred to as megamorphic. In this state, efficient polymorphic dispatch is not possible, since too much time would be consumed walking the cache entries and comparing them to the new receiver. Megamorphic call sites often revert back to an unoptimized vtable-based dispatch mechanism [16].
Chapter 5

Kale Language

5.1 Overview

Kale is a strongly typed programming language with structural interfaces that has been designed to prototype structural typing on the JVM using invokedynamic. Many design elements of Kale are inspired by the Google Go [17] programming language.

5.1.1 Types

Type declarations in Kale are analogous to classes in Java. Types can be declared with zero or more fields and methods, all of which are annotated with type names. Listings 5.1 and 5.2 show how Java syntax maps to Kale syntax.

```
package test;

class Person {
    int age;
    String name;
    String getName() {
        return this.name;
    }
}
```

Listing 5.1: A Java code example

```
package test

type Person {
    age int;
    name string;
    getName() string {
        return this.name;
    }
}
```

Listing 5.2: A Kale code example
Kale using trailing type annotations instead of the leading type annotations of most C-derived languages. Google Go’s development team found this allows more complex declarations to be read left to right [18] instead of the “spiral pattern” of C [19]. Regardless of the small syntactical differences, both Listing 5.1 and 5.2 compile to the same bytecode.

Structurally typed interfaces are declared with the `interface` keyword and can only contain method declarations – no method bodies. Listing 5.3 defines an interface `A` that is implemented by type `T`. Note that if the return type of a method is omitted, `void` is assumed.

Kale, unlike Java, allows functions to be declared outside the scope of a class. Listing 5.4 shows a legal Kale program with only a main function declared.

### 5.1.2 Operators

Kale implements a simplified version of operator overloading. Listing 5.5 demonstrates a `Vector` type that has a `+` operator.
Operators are made possible by Kale’s relaxed tokenization rules. The only special lexeme types are control characters like parentheses, braces, and brackets, or punctuation like semi-colons, commas, and full stops. All other lexemes, even arithmetic operators, are considered symbols. Operators are detected in the parser by an occurrence of three consecutive symbols, in which case the first and third symbols are considered the operands and the second symbol is considered the operator. Usage of the operator defined in Listing 5.5 is shown in Listing 5.6.

Operator invocation is constructed as a chain of function invocations, so another way of writing \( v_1 + v_2 \) would be \( v_1.+(v_2) \) and \( v_1 + v_2 + v_3 \) would be \( v_1.+(v_2).+(v_3) \). Parentheses can alter evaluation order, turning \( v_1 + (v_2 + v_3) \) into \( v_1.+(v_2.+(v_3)) \).

This ignores the usual order of operations for basic arithmetic, but makes reasoning about operator use in complex types like a `Vector` or `Point` more straightforward.

### 5.2 Compilation

The Kale compiler accepts Kale source and generates JVM bytecode as output. The compiler is implemented in Java and uses a custom lexer and recursive descent parser to build an abstract
syntax tree (AST). JVM bytecode is emitted directly from the AST through calls to the ASM bytecode generation library [20], and can be loaded directly into a ClassLoader and executed or packaged into a jar file.

Since Kale programs differ slightly from Java in structure and content, some conversions must be applied in order to map Kale programs to class files. First, since Kale allows a broader range of characters in symbols compared to Java, symbols used for class and method names must be converted to an equivalent unqualified name as defined by the JVM specification [5, 4.2.2]. Additionally, functions defined outside the scope of a class within a package must be packaged inside of a class. For this reason, a _functions class is generated within the declared package that contains all package-level functions.

5.2.1 Invocations

Since Kale uses a mix of concrete types and structural interfaces, invocations generate a mix of invokevirtual and invokedynamic instructions. Listing 5.7 defines an interface, Gettable, and a type that implements that interface, Stream. In the main() function a new instance of Stream is created, and passed as a parameter to callGet(Gettable). The invocation g.get() generates an invokedynamic call site, since it is being invoked against an interface. All other invocations, including the call to internal() within the Stream class, use an invokevirtual instruction.
It is tempting to simply replace every invocation with an `invokedynamic` instruction to simplify code generation and allow for maximum flexibility. However, the MethodHandle lookup and CallSite construction still have a non-zero cost compared to the JVM internal linking which requires no reification or simulation.

Listing 5.8 shows the `invokedynamic` instruction that would be generated from `g.get()` in Listing 5.7. This invocation specifies the Kale runtime library static method that will provide a call site and takes advantage of the ability to embed parameters directly into the call site. By specifying the name of the interface as a String parameter the bootstrap method can verify whether a given receiver conforms to that interface at runtime.

```
invokedynamic  get(Ljava/lang/Object;)V [ // handle kind 0x6 : invokevirtual
kale/runtime/Bootstrap.bootstrap(
    (Ljava/lang/invoke/MethodHandles$Lookup;
    Ljava/lang/String;
    Ljava/lang/invoke/MethodType;
    Ljava/lang/String;
    )Ljava/lang/invoke/CallSite;
```
Listing 5.9 shows the corresponding bootstrap method declaration that will be called by the JVM runtime in order to provide a call site for the interface call site.

```java
public static CallSite bootstrap(MethodHandles.Lookup caller,
                                  String name,
                                  MethodType type,
                                  String targetType)
{
    Class<?> targetClass = Class.forName(
                                targetType,
                                true,
                                caller.lookupClass().getClassLoader());

    InterfaceCallSite ics = new InterfaceCallSite( targetClass,
                                              name,
                                              type );

    ics.rootHandle = MethodHandles.
                     .insertArguments( ics.rootHandle, 0, ics )
                     .asVarargsCollector( Object[].class )
                     .asType(type);

    ics.setTarget( ics.rootHandle );

    return ics;
}
```

Listing 5.9: Interface call site runtime bootstrap method

The role of the bootstrap method is to populate the initial state of the call site with enough information to upgrade itself in response to new receiver types as they are encountered. The
A MethodHandle pointing to call is set as the root of a MethodHandle tree similar to the tree depicted in Figure 4.3. Each subsequent invocation of call indicates that all cached receiver tests (the guard tests higher in the MethodHandle tree) have failed – a cache miss – so a new guard testing for the receiver type is set as the new root of the tree with its fallback set to the current root. Each InterfaceCallSite retains a cacheDepth integer which tracks how many cached types the call site has in its MethodHandle tree. Once this exceeds a preset threshold – currently five – the call site is marked as megamorphic by setting InterfaceCallSite.cacheDepth to -1. After that, the cache is discarded and all subsequent invocations perform a reflective lookup against the receiver.
5.3 Performance

Structural invocation through a `MethodHandle` cache was benchmarked against several other invocation techniques to test its performance. Three types – A, B, and C – are defined within both Kale and Java with a single no-argument method `m` that returns a string constant. Within Java, all three types declare that they implement interface I which also defines method `m`.

The baseline performance metric is an `invokeinterface` instruction, which is heavily optimized and can take advantage of the limited number of implemented types inherent to explicit interface implementation as discussed in Chapter 4. Next, two Core Reflection API-based implementations, `invokeReflection` and `invokeCachedReflection`, were implemented to compare with the findings of Dubochet and Odersky’s findings in their Scala-oriented structural typing implementation [12]. `invokeReflection` does a full method lookup from the receiver `Class` object on each invocation, and `invokeCachedReflection` implements a simple cache of the resolved `Method` objects. Additionally, a generative technique, `invokeStaticInline`, which was also discussed as an option by Dubochet and Odersky, was implemented that defines an inline cache that is statically defined at compile time with a fixed set of types. The `invokeStaticInline` approach is completely impractical in the real world, but it is important because it is written in Java, as opposed to the native implementation of `invokeinterface`. If the `MethodHandle` runtime optimizer was ”sufficiently smart”, it would be able to generate equivalent native code (see Listing 5.11). Finally, `invokeMethodHandle` is the `MethodHandle`-based adaptive polymorphic inline cache.
For each approach, an instance of \texttt{A}, \texttt{B}, and \texttt{C} were created as invocation targets. Then 100000 iterations of invocations were performed, alternating \texttt{A}, \texttt{B}, and \texttt{C} as parameters to a function that accepted a parameter of type \texttt{I}, where an invocation was performed against method \texttt{m}. The benchmark was performed on a 3.20 GHz 64-bit AMD Athlon II X3 450 in Windows 7 running JVM 1.7.0.07 with arguments \texttt{-server -XX:+AggressiveOpts}. The execution times, averaged over three runs of a cold JVM, are displayed Table 5.1.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>invokeinterface</td>
<td>4</td>
</tr>
<tr>
<td>invokeReflection</td>
<td>296</td>
</tr>
<tr>
<td>invokeCachedReflection</td>
<td>45</td>
</tr>
<tr>
<td>invokeMethodHandle</td>
<td>22</td>
</tr>
<tr>
<td>invokeStaticInline</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5.1: Execution time of interface invocations.

Over multiple test executions, \texttt{invokeMethodHandle} consistently outperformed the reflective inline cache by a factor of two. \texttt{invokeMethodHandle} was still not as fast as \texttt{invokeinterface}, but that is a highly optimized native implementation that has been developed over several years. \texttt{invokeStaticInline} is the most telling of the results, since it represents a theoretical best case performance. The runtime optimizer for method handles is new in JDK 7, so it is possible that it will gain performance as it matures relative to the other JVM runtime code generators.
Chapter 6

Conclusion

A structurally typed programming language was developed to target the new JVM `invokedynamic` bytecode instruction and runtime support libraries. After the compiler was implemented, the resulting compiled programs using `invokedynamic` were benchmarked against existing solutions, including the `invokeinterface` instruction and inline caches constructed using the Core Reflection API. The `invokedynamic` solution outperformed the Core Reflection API-based solutions by a factor of two, but could not match the performance of the native `invokeinterface` instruction.

The `invokedynamic` standard library API is powerful and the promise of abstraction without penalty of indirection is compelling. The potential applications of `invokedynamic` and method handles in general are widespread, including the possibility of pointers on the JVM and specializing data structure execution paths as runtime conditions change. As the method handle runtime optimizer is improved in future versions of the JVM, `invokedynamic` could become a viable general purpose replacement for the other invocation instructions.
Bibliography


