Bytecodes meet Combinators: `invokedynamic` on the JVM

John R. Rose

Sun Microsystems

john.rose@sun.com

Abstract

The Java Virtual Machine (JVM) has been widely adopted in part because of its classfile format, which is portable, compact, modular, verifiable, and reasonably easy to work with. However, it was designed for just one language—Java—and so when it is used to express programs in other source languages, there are often "pain points" which retard both development and execution. The most salient pain points show up at a familiar place, the method call site.

To generalize method calls on the JVM, the JSR 292 Expert Group has designed a new `invokedynamic` instruction that provides user-defined call site semantics. In the chosen design, `invokedynamic` serves as a hinge-point between two coexisting kinds of intermediate language: bytecode containing dynamic call sites, and combinator graphs specifying call targets. A dynamic compiler can traverse both representations simultaneously, producing optimized machine code which is the seamless union of both kinds of input. As a final twist, the user-defined linkage of a call site may change, allowing the code to adapt as the application evolves over time. The result is a system balancing the conciseness of bytecode with the dynamic flexibility of function pointers.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features – Procedures, functions, and subroutines, polymorphism, control structures; D.3.4 – Optimization, Code generation.

General Terms Performance, Standardization, Languages

Keywords Bytecode, combinator, method invocation, invokedynamic, dynamic compilation

1. Bytecode Preliminaries

The Java™ Virtual Machine specification [Lindholm99] defines a classfile format which serves as an intermediate language. As described in early accounts [Gosling95], this format is a bytecode-based intermediate language designed to be compact, amenable to both interpretation and translation, portable, statically verifiable (rich in types), and symbolically linked (free of addresses or offsets). These last two qualities allow JVM modules (classfiles and JAR files) to be safely and robustly composed.

The classfile format, along with the rest of the Java technology, has been extraordinarily successful; by some metrics [TIOBE09, Ohloh09] Java has been the most or second-most popular programming language, in a broad range of communities, for most of a decade. It is likely that a large fraction of *all* instructions executed by computers worldwide have been specified via Java bytecode.

The symbolically resolved names that occur within a class file are classes, methods, and fields. The resolution of these names is governed by complex rules. For example, class name resolution is partially programmable via *class loader* objects. These rules are based on and designed for the Java language itself [Steele96], and are a fixed behavior of the JVM.

References to fields and methods are strongly typed, with no variance allowed between a definition and its uses. That is, matching is exact, for both member names and type signatures. Member lookup and matching of names and types is performed once only as a symbolic reference is *linked*, and specifically when it is *resolved* ([Lindholm99] 5.4.3). This happens before the first time a particular member reference instruction executes.

Every field or method reference starts with a scope type (a class or interface), and proceeds up the Java type hierarchy to find the addressed member. Java's access control rules are also applied. If the member is not statically defined and accessible among the supertypes of the initial scope, the instruction fails to link, and never executes. There is no provision for fallback logic like the `doesNotUnderstand:` protocol defined in Smalltalk [Goldberg83].

Method invocation is the principal mechanism for composing software modules (classes, packages, applications) that access each other within the JVM. The four method invocation instructions in the JVM correspond directly to different kinds of Java method calls. All of them consult a scope for a named method with a given type signature. Figure 1 summarizes them.

mnemonic	byte	scope type	receiver?	dispatch?
`invokestatic`	`B8`	class	no	N/A
`invokespecial`	`B7`	class	yes	no
`invokevirtual`	`B6`	class	yes	yes
`invokeinterface`	`B9`	interface	yes	yes

Figure 1. Summary of JVM invocation bytecodes.

Except for `invokestatic`, each invocation treats an extra leading argument as the receiver object, and statically types it to the scope type. The receiver, or any other reference argument, may have a dynamic type that is a subtype of the static type used for linkage. An invocation that performs dispatch does so only on the receiver's class. There is no multiple dispatch [Musch08] in the JVM. Overloaded function calls in Java are resolved at compile time and appear to the JVM as having fixed type signatures.

As a final constraint, the names of types and members must avoid certain 7-bit ASCII characters, such as forward-slash, period, and semicolon. Schemes to relieve this minor restriction exist [Rose08], but are beyond the scope of this paper.

2. The Multi-Language JVM

These rules, while perfect for supporting Java programs, are a mixed blessing to implementors of other languages. Before we discuss difficulties, it is well to note that implementors are drawn by the well-known strengths of the JVM platform. These include reflection over metadata, optimizing compilers, efficient garbage collection, scalable concurrency, wide distribution, and robust security. Also, the JVM is surrounded by a rich ecosystem of libraries, applications, tools, and communities. A decade or more ago, language implementors often coded their own runtimes. Now many have chosen to reuse complete VMs like the JVM or Microsoft's CLR [Meijer00]. In recent years, the choice to reuse the JVM has been made many times. By one count [Tolksdorf09], the number of JVM languages is presently at 240.

The difficulties of adapting the JVM to languages other than Java have long been noted. Architectural difficulties or "pain points" for language implementors can involve requirements for structured value types (specifically as return values), open classes, parameterized types, method pointers, tail calls, or continuations of various species. The Da Vinci Machine project [DaVinci09] aims to investigate such pain points and create prototypes of well-designed solutions.

2.1 Multi-Language Method Calls

Many pain points appear as language implementors attempt to match the method or function invocation rules of their languages to those of Java. Let us examine this problem in detail.

A language-specific notion of method or function invocation can often be analyzed into a linkage from a method *use* (call or invocation) to a method *definition*, specified in the following JVM-like terms:

- a method name (N^u for the use, N^d for the definition)
- a scope in which to seek the method (S^u) or define it (S^d)
- zero or more argument types ($A^u_{(i)}$, $i < |A^u_{(*)}|$; $A^d_{(j)}, j < |A^u_{(*)}|$)
- zero or more return value types ($R^u_{(i)}$, $R^d_{(j)}$)
- a symbolic method reference $M^u = \langle S^u, N^u, A^u_{(*)}, R^u_{(*)} \rangle$
- access control contexts, if any (C^u, C^d, typically classes)
- access permission $P^d \in$ *{public, private, protected, package}*
- a method definition M^d consistent with the $A^d_{(*)}$ and $R^d_{(*)}$

Let us also assume, for now, that there is at most one return type, and that the argument and return types are all JVM references or one of the eight JVM primitive types.[1] We will also assume a classic pattern of control transfer consisting of a single call followed by a single return or throw.[2] Typically, C^u will be a class whose bytecode contains the reference M^u.

In these terms, languages often need some of the following degrees of freedom, beyond what Java and the JVM support:

a) N^u might differ from N^d (or M^d might be nameless)

b) N^d might be a symbolic entity other than a Java identifier

c) S^u might differ from, and not be a subtype of, S^d

d) S^u and/or S^d might be something other than a class or interface

e) two types $A^u_{(i)}$, $A^d_{(i)}$ or $R^u_{(i)}$, $R^d_{(i)}$ might differ pairwise

f) types may differ in number: $|A^u_{(*)}| \neq |A^d_{(*)}|$ or $|R^u_{(*)}| \neq |R^d_{(*)}|$

g) the class C^u might not have access permission to P^d in C^d

Examples requiring such extra flexibility are Scheme top-level bindings, Common Lisp packages, Smalltalk classes, JavaScript scopes, Groovy overloaded calls, Ruby optional arguments, or C# `friend` references, respectively.

Beyond these degrees of freedom, linking a language-specific symbolic reference M^u to a desired target method M^d also requires more options than the JVM natively implements. Any of the following situations may arise:

h) the linked-to entity M^d might be an object instead of a method

i) the initial linkage to M^d may require specialized logic

j) the call site may require occasional relinking to some $M^{d'} \neq M^d$

k) the call site may need to be reset to an unlinked state

l) the call site may need to perform language-specific dispatch

m) the call site may need to emulate a bytecode instruction

Usage examples are global names bound to closures, any language with complex global scoping rules, languages with mutable classes, reuse of a precompiled script, arithmetic on dynamically-typed operands, and calls out to Java APIs, respectively.

Finally, the language may mandate conversions of arguments, return values, or exceptions. If the conversions cannot be precompiled into the call site (e.g., because they happen after type tests), they must be somehow adjoined or "mixed into" M^d. Usage examples include conversions between numeric, sequence, or string formats, and wrapping or unwrapping of exceptions.

2.2 Reified Methods have Simulation Overheads

Since Java is a general-purpose language, these extra degrees of freedom or options can be simulated by a suitable object-based API, reifying M^d as a method m on an object s. To emphasize the extra representational layer, we will call s a *method simulator object*, and m a *simulator method*.

The call site information M^u may also need reification, in part or whole. In the latter case we call the resulting object c a *call site simulator object*. Call sites containing language-specific linkage state or dispatch caches generally require such simulators.

In JRuby [Nutter09] *s.m* is named `DynamicMethod.call` and c's type is `org.jruby.runtime.CallSite`. A detailed account is [Lorimer09]. Clojure [Hickey09] uses the simulator method `clojure.lang.IFn.invoke`. In Jython [Hugunin09] method simulation is done by the root type `PyObject`. From the JVM's own Core Reflection API, the simulator method `java.lang.reflect.Method.invoke` is a building block for many language implementations.

When the method call site and/or target method definition is reified as an explicit Java object, its behavior becomes malleable relative to hardwired JVM calls. The explicit objects are coded to simulate not only the JVM's built-in linkage and calling behaviors, but also language-specific extensions and special features.

Malleability can, however, come at the cost of serious execution overhead and complexity. The problem is that simulators do not express, directly and unambiguously, the sort of method invocation operations that the JVM is designed to optimize. Specifically, we can recognize the following pain points:

1. **methods only:** The JVM cannot bind a call site directly to an arbitrary object M^d, so indirection is needed, which requires extra bytecode and Java data structures.
2. **non-constant loads:** Extra indirections can defeat attempts by the JVM to predict and inline call targets.

[1] Proposals to remove these restrictions, such as [Rose04], are beyond the scope of this paper.

[2] Again, tail calls [Schwaig09], continuations [Stadler09], coroutines, etc., are a problem for another time.

3. **constant type restrictions:** The initial reference *s* or *c* of the indirection chain must be maintained as a static variable, thread-local, or hidden argument. Alternatively, if *s* and *c* are shared constants (or *m* is static), the system cannot use optimization techniques such as inline caches (described in section 5 below) which rely on per-call-site state.
4. **polluted profiles:** The simulation mechanism is usually factored such that one logical method call M^u requires several JVM-level method calls $s.m \rightarrow m_1 \rightarrow \ldots$ If this is so, some of the m_i execute shared code, through which the execution traces of many M^u must pass. Aliasing many call traces onto one shared "call interpreter" routine (e.g., m_1) can defeat attempts to profile the logical calls and predict ultimate targets.
5. **failed inlining:** Multiple-level method calls may also overtax "inlining budget" heuristics in the compiler, thus causing logical call sites to be optimized much more poorly.
6. **obscured optimizations:** Extra indirections and method calls can obscure the shape of language-specific optimizations at the call site. The language runtime may try to transform a call site to build in a fast path, but it will be wasted effort if the improvement is obscured in the shape of the executable code.
7. **awkward code:** If a dynamic compiler cannot inline enough of the logical call site due to any of the above reasons, it cannot easily organize the machine code to enable the hardware (caches, pipelines, prefetch logic, etc.) to operate as designed.
8. **simulator signatures:** Simulated calls usually drop some of the signature information, adapting $\langle A^u_{(*)}, R^u_{(*)} \rangle$ to a simplified signature $\langle a^s_{(*)}, r^s_{(*)} \rangle$for *s.m*. Let us call these simplified signatures *simulator signatures.* For example, the $a^s_{(1)}$ might be widened to `Object`, or converted from a primitive `int` to a boxed `Integer`. This adds complexity to the system.[3]
9. **representation shifts:** Argument and return value conversions are required to match simulator signatures. Basic conversions include casting, boxing, and "varargs" manipulations. Such complexity can waste optimization work on simple data movement. Language-specific conversions may also occur, but the ones blamable as "waste" are those which express the same logical value in multiple JVM-level representations.
10. **rich target signatures:** Even if the $\langle A^u_{(*)}, R^u_{(*)} \rangle$ are kept simple and aligned with $\langle a^s_{(*)}, r^s_{(*)} \rangle$, a call to a M^d within a Java API requires conversions to match the $\langle A^d_{(*)}, R^d_{(*)} \rangle$. This can manifest as a need to generate thousands of "Java invokers" which adapt all possible Java M^d to corresponding *s.m* calls.[4]
11. **metadata constants:** Metadata about the call site may need to be reified and conveyed to *s.m*. This is most often the case with the name N^u, which usually appears as an extra argument.
12. **call site identity:** The creation of a call site simulator object *c* is not controlled by the JVM. This means that inlined or optimized copies of a single logical call site will share a common simulator *c*, even when it would be profitable to split *c* into multiple copies c', c'', and so on. JVMs which use inline caches for native invocations rely on the ability to split native call site states in this way.
13. **nonstandard infrastructure:** The reification and simulation techniques invented by language implementors occur in profuse, even dizzying variety. This variety makes it unlikely that the JVM will reliably recognize them as intended, or that JVM vendors will invest serious effort in optimizing them.

3. Simulators Revisited: `invokedynamic`

A new invocation mode, embodied by the `invokedynamic` bytecode instruction, is now being defined for the JVM to address the pain points described in the previous section. The Java Community Process, the body which oversees Java-related standards, has chartered the JSR 292 Expert Group to design this instruction. The multi-year design effort has led the JVM into new territory. Beyond our initial expectations, the creation of the new invocation mode has also forced the creation of a new, more direct, form of method simulator object called a *method handle*. In a nutshell, function pointers have arrived in the Java virtual machine.

First let us describe `invokedynamic`. It is a 5-byte instruction similar to `invokeinterface`, but it has no built-in scope, receiver, or dispatch behavior. Figure 2 is the final line of the table in Figure 1.

mnemonic	byte	scope type	receiver?	dispatch?
`invokedynamic`	`BA`	none	N/A	N/A

Figure 2. Summary of `invokedynamic` bytecode.

Because the new instruction lacks a scope type S^u, its symbolic reference is limited to a simple name-and-type *descriptor* $D^u = \langle N^u, A^u_{(*)}, R^u_{(*)} \rangle$. The reference occupies two bytes. The final two bytes of the instruction format are required to be zero; they are reserved for future use. An instance of an `invokedynamic` instruction is called a *dynamic call site.*

Like all `invoke` instructions, `invokedynamic` can be linked lazily, as late as the first execution of each dynamic call site. The JVM consults a locally defined *bootstrap method* to reify the call site as a simulator object *c* (see next subsection). This object *c* is then associated with the dynamic call site. The association is persistent within the JVM, until unlinking is requested. The call then determines its target M^d simply by picking up a method handle *s* from a field of *c*, and running a simulator method *s.m*.

The dynamic call site can be executed any number of times. The language runtime is free to manage the call site by updating the target *s* from time to time, and/or unlinking *c* to force recreation of a new call site simulator (re-reification, as it were).

In general, the method handle *s* may be viewed as the root of a small graph of combinators whose leaves are direct references to other bytecoded methods. We will examine the details of method handle structure in section 4.

Because *c* is persistent, and its target *s* is usually stable, a JVM dynamic compiler can elect to constant-fold the values of *c* and *s*, and inline any code or executable logic that *s* refers to. Before we look more closely at this inlining process, let's fill in more details about the reified call site *c*.

3.1 Bootstrapping Dynamic Call Sites

Before an unlinked dynamic call site is executed, the JVM calls the site's bootstrap method. Each class C^u containing dynamic call sites registers its own bootstrap method by calling a system method from within the class initializer. A bytecode compiler

[3] In the Core Reflection API, the simulation signature $\langle a^s_{(*)}, r^s_{(*)} \rangle$ is always $\langle$`Object[]`, `Object`$\rangle$. JRuby has about 8 simulator signatures, and Clojure has about 21.

[4] In HotSpot, reflection is currently implemented like this, with a bytecoded invoker generated on demand for each method.

which generates `invokedynamic` instructions is responsible for generating code to register a bootstrap method for each such C^u.

Callee methods M^d are not yet determined at link time, and thus have no effect on bootstrapping. Any stacked arguments are also ignored during linking. No receiver argument is distinguished. Bootstrapping is thus totally unlike Smalltalk's `perform:` or `doesNotUnderstand:` protocols.

The bootstrap method is passed parameters identifying all the statically determined symbolic information in the call site, specifically C^u and $D^u = \langle N^u, A^u_{(*)}, R^u_{(*)} \rangle$. The name N^u can be an arbitrary string, subject only to the JVM's rules for method name formation. The JVM never interprets the contents of this string, so language runtimes may use it for any purpose to help determine the intended target method M^d, providing generalizations (b) and (i) in section 2.1 above. In particular, part of this string is sometimes used to encode a logical lookup scope S^d, providing a path to generalizations (c) and (d) in section 2.1 above.

The bootstrap method creates a call site object c and returns it to the JVM. The bootstrap method may be viewed as a call site factory. The object c must be a `java.dyn.CallSite` or a subclass thereof. The subclass may be a user-defined class that embodies language-specific behavior for processing the call. The JVM persistently associates c with the specific `invokedynamic` instruction that required the bootstrap, until and unless it is unlinked by a request from the language runtime.

Further interactions with the JVM at that dynamic call site are mediated through the call site object. Bootstrapping provides the "hook" for language-specific logic to set up the call site, but afterwards all calls will run by a more direct path, described next.

3.2 Making the Call

When a dynamic call site is executed (after it is linked), the JVM extracts from the call site object c a target method reference s. This reference is a method handle, which we will describe in detail in section 4. In the reference implementation, s is loaded from a private field named `CallSite.target`. The key point is that s has the potential to refer directly to any JVM method whatever.

The JVM transfers control directly from the dynamic call site to the simulation method $s.m$, which is named `invokeExact`. The simulation method may jump immediately to an ultimate target method M^d or it may first perform argument transformations necessary to adapt to $A^d_{(i)}$. This flexibility in method handles provides generalizations (e) and (f) in section 2.1 above.

Because s is intrinsically anonymous, its ultimate target M^d can have a name N^d that is unrelated to the name N^u in the dynamic call site. This provides generalization (a) in section 2.1 above.

3.3 Static Type Checking, Correct by Construction

The static types $A^u_{(i)}$ of the outgoing stacked arguments and incoming return value $R^u_{(i)}$ must pairwise agree exactly with the static types of the stacked parameters $a^s_{(i)}$ accepted by the method handle and return values $r^s_{(i)}$ produced by it. Specifically, the method type signatures used by the JVM must match between the dynamic call site and the simulator method. This allows the JVM to neglect all data conversions when calling the method handle.

The alert reader will be wondering how a method handle can declare an `invokeExact` method for every signature. This will be explained below; it is enough to say here that all possible JVM signatures, with and without primitive types, are supported by both dynamic call sites and by method handle entry points.

The exact match of the method handle's signature to the call site is assured as an invariant on the `CallSite` object c, so the JVM does not need to check signatures on every dynamic invocation. Any attempt to create a call site with a mismatched target type will cause a linkage failure error. The effect on global type integrity is as if the JVM verifier checks the method signature at a dynamic call site when its initial target method s is specified, and also whenever (if ever) it is later relinked to another target s'.

3.4 Dynamically Changing the Call

A dynamic call site may have its target updated after bootstrapping. The `CallSite` API methods `getTarget` and `setTarget` are used to manage the target. The allowed side effects are carefully limited to preserve type correctness. The new target must have the same type, or an exception is thrown and the call site is left unchanged. This mutability provides generalization (j) in section 2.1 above, and allows dynamic call sites to respond quickly to changes in application configuration.

More crucially, a language runtime may modify a call site in response to observed operand types in order to optimize it locally. As section 5 below explains, the classic inline cache optimization keeps track of operand types at a single call site, and reconfigures it over time into neutral, monomorphic, and megamorphic states, responding to the needs of the application. Mutable call sites allow language runtimes to compose their own inline caches. Also, many JVM implementations already use inline caches internally to optimize `invokevirtual` and `invokeinterface`. In those JVMs, `invokedynamic` can be supported by some of the same infrastructure that underlies the older instructions.

Mutable calls have tricky interactions with the Java memory model. Within the same thread, the next call after a use of `setTarget` will observe the new target. Other threads may observe the new target later, because a call site's target behaves, with respect to global memory, like a regular, non-volatile heap variable.

Modern JVMs generate optimized code concurrently with application execution. An optimized version of a dynamic call site might inline all the way through the target method handle and into the ultimate target method M^d. Changing the target method handle might cause such code to be patched or discarded.

Clearly if this happens many times at one dynamic call site, the JVM's heuristics should identify the call site as *megamutable*, and generate target method linkage code which is slower, more decoupled, and more general.

There is a second possible state change on dynamic call sites, and that is to discard them by marking them unlinked. This forces a new bootstrap cycle on the next execution of the instruction. Unlinking operations are strongly serialized across the whole JVM. They are supposed to be rare, and JVMs may not optimize such state transitions well, but if there is a problem with inter-thread latency, unlinking is a "big hammer" for resetting call sites. With this fallback, the normal case code for `invokedynamic` can avoid any need for volatile memory references.

No other changes are possible to linked call sites or their target method handles. In particular, method handles are pure values, whose structure cannot be changed after creation. This is why the compiler will be able to inline them as well as it inlines bytecode.

3.5 Summary of `invokedynamic`

The design just sketched is complete in outline. Here are the key principles we have followed:

- Provide built-in simulator types for call sites and methods.
- Make the simulators be as similar as possible to native call sites and methods, especially in performance characteristics.
- Try to keep language logic on a par with JVM linkage logic.
- Reify the language logic, but keep it out of the fast path.
- Support all JVM method signatures, not just a "dynamic" subset. This feature may be called *signature polymorphism*.

4. Method Handles

So far we have been vague about the structure of method handles. In fact, before JSR 292, Java did not provide a data structure which corresponds to a direct reference to an arbitrary method. In order to design `invokedynamic`, however, it became clear that we needed a data structure which can answer the question, "to what target is this dynamic call site bound?"

The closest pre-existing candidate might have been `Method` from the Core Reflection API, but the reflective `invoke` method has very high simulation overheads, such as a full access permission check on every method call. Reflective methods are also symbol table entries, since they supply a full account of annotations, generic types, and throws. All that structure would be expensive overhead to dynamic call sites, where the only thing that matters is fast invocation through function pointers.

Therefore, the target of a dynamic call site is described with a new class, `java.dyn.MethodHandle`.

The design principles for method handles complement those for `invokedynamic`. The basic idea is to supply "JVM plumbing fittings" in enough variety that dynamic call sites can serve as flexible connection points between callers and callees. Here are the specific principles:

- Provide direct invocation of JVM methods.
- Do not hardwire functions proper to language logic, such as name lookup, access checking, exception transformation, etc.
- Allow calls to all JVM methods, including private ones.
- Be polymorphic across all JVM signatures.
- Provide uniform access to basic JVM operations, including the `invoke` instructions.
- Support function composition, in the mathematical sense.
- Support argument adaptation via casting, boxing, etc.
- Support partial argument application, as for closures.

Every method handle has a specific type, a signature under which its `invokeExact` method can be invoked. If bytecode stacks arguments and attempts to call `invokeExact` under a different signature, an exception is thrown, as section 4.2 describes.

This type can be queried by the `type` method. The result is an instance of `MethodType`, an immutable value encoding a series of argument types and zero or one return types. Other than a type and an invocation point, a method handle's structure is opaque.

4.1 Direct Method Handles

The simplest form of method handle is the *direct method handle*, which is a direct reference to a Java method. Invoking a direct method handle is equivalent to invoking the method it points to. Here is a simple example which calls `System.out.println`:

```
MethodHandle println = MethodHandles.lookup()
    .findVirtual(PrintStream.class,
      "println", MethodHandles.methodType(
      void.class, String.class));
println.<void>invokeExact(
    System.out, "hello, world");
```

Figure 3. Creating and using a direct method handle.

Several things happen in this example. The class C^u will both create and call a method handle object. First, the `lookup` call performs an access check on C^u (via a stack walk) and builds a capability object of type `MethodHandles.Lookup`, which represents the permissions of C^u. This step is typically done once per class, storing the `Lookup` object in a private static final.

Next, a call to a factory method obtains a `MethodType` representing a signature taking a string argument and returning no value.

Finally `Lookup.findVirtual` verifies that `PrintStream` contains a `println` method of the required signature and creates a method handle for it. Because the target method is virtual, the handle takes an extra leading argument for the receiver. It is of type `PrintStream`, as Figure 4 shows.

```
MethodType mt = println.type();
assert mt.parameterType(0)==PrintStream.class;
assert mt.parameterType(1)==String.class;
assert mt.returnType()==void.class;
```

Figure 4. Method handle type, with inserted receiver argument.

Like `invokedynamic`, and unlike Core Reflection, method handles do not treat receivers specially. When a receiver is present, it is simply the first argument. This leading receiver argument appears like any other argument in the method handle's type.

Thus, the `println` handle, viewed as a simulator, has a signature $\langle a^s_{(0)}, a^s_{(1)}, r^s_{(*)}\rangle$ of ⟨`PrintStream`, `String`, `void`⟩, which matches `println`'s entry point signature $\langle A^d_{(0)}, A^d_{(1)}, R^d_{(*)}\rangle$.

The odd-looking type parameter `<void>` in the `invokeExact` call is a proposed extension to the Java language. This parameter, plus the static types of the arguments, determines the simulator signature as it appears in the bytecode.

Method access checks are based on a method handle's creator, not its caller. In the example above, there is just one class, but if the creating class C^c differs from the calling class C^u, then any access checks on the method are done relative to C^c, not C^u. This is different from the Core Reflection API, which may reflect a private method, via `Class.getDeclaredMethods`, from any class C^c, but on every call performs an access check against every C^u. Of course, doing so slows down reflective invocation.

More precisely, a class C^c can form a method handle on any method it is allowed to call, specifically including any of its own private methods ($C^c = C^d$) and methods in the same inner class scope (C^c, C^d contained in a common package member). It is assumed that C^c will share such private handles only with classes C^u that it trusts enough to call the methods. In effect, creator-based access checks allow C^c to *begin* a call that some combinator C^u will later *finish*. This supports generalization (g) in section 2.1.

Direct method handles can emulate all the `invoke` bytecode instructions, plus reflective calls. The distinctions are implied via the `Lookup` method as follows:

- `invokestatic` is emulated via `Lookup.findStatic`
- `invokespecial` via `Lookup.findSpecial`
- `invokevirtual` via `Lookup.findVirtual`
- `invokeinterface` also via `Lookup.findVirtual`
- `reflect.Method.invoke` via `Lookup.unreflect`

Note that two direct method handles can refer to the same method but differ in their behavior, in that one performs receiver-based dispatch on its first argument, while the other does not. The first case is far more usual. See generalization (m) above.

As with the actual `invokespecial` instruction, the `findSpecial` operation requires extra permission checks. This feature allows dynamic languages to extend inherited Java methods as if via the "`super`" syntax.

4.2 Invoking Method Handles

Every method handle's principal entry point is `invokeExact`, a public abstract method. Uniquely, the signature of this method varies from method handle to method handle, but always corresponds exactly to the method handle's own type.

The bytecode to call this method is familiar. The method handle is stacked, and becomes the receiver of an `invokevirtual` instruction of `MethodHandle.invokeExact`. Zero or more additional arguments are stacked, and the JVM verifier ensures that they are consistent with the instruction's method signature. This signature must also exactly match the method handle's own type. If this match fails, the JVM throws an unchecked exception named `WrongMethodTypeException`. No conversions occur.[5]

The current JSR 292 draft has another invocation mode for method handles which includes argument and return type conversions. It appears as another virtual method beside `invokeExact`, named `invokeGeneric`. The two modes are distinguished as *exact invocation* versus *generic invocation*. For any given method handle, `invokeGeneric` may be invoked under exactly the same signature as `invokeExact`, with the same effect. However, if the calling signature $\langle A^u_{(*)}, R^u_{(*)}\rangle$ differs from the method handle's simulation signature $\langle a^s_{(*)}, r^s_{(*)}\rangle$ the JVM will make up the difference between signatures (perhaps with extra execution cost), by applying cast, box, and unbox operations as needed. The rules for doing this are beyond the scope of this paper.

Generic invocation makes method handles easier to use. With only exact invocation, method handles could not conform to the usual variance rules for static typing of functions, and thus could not represent function pointers in many languages.

4.3 Adapter Method Handles

A method handle *m* of type *t* can be adapted to a new method type *t'* by creating an *adapter method handle*, a wrapper *a* of the desired new type, which delegates to the original target *m*.

If *t* and *t'* have the same arity, the library method `MethodHandles.convertArguments` accepts *m* and *t* and returns such an adapter *a*. As one might expect, an exact invocation of *a* under the signature of *t'* is equivalent to a generic invocation of *m*, also under *t'*. This library method can be thought of as a sort of "coping combinator", for dealing with the wide variety of method type signatures the JVM offers. In particular, it can represent any method handle as a new method handle that operates only on `Object` arguments and return values.

It has been pointed out [Öhrström09] that if there were no exact invocation mode, and method handles were to support *only* generic invocation, there would be no need for `convertArguments`; certain other simplifications would also become possible. The rejoinder to this is that on-the-fly argument conversion is not a feature of native JVM method calls, and so should be an optional feature of method handles. Such conversions are very useful at times. Because there is a choice of modes, programmers can elect at other times to avoid conversion costs.

Other argument list adaptations are possible, to reorder, duplicate, drop, collect, and spread arguments. Adapters can also apply user-defined conversions to arguments. All adapters are immutable and opaque. Figure 5 is a summary of the standard adapter combinators provided in the JSR 292 draft design.

combinator	argument transformation
`convertArguments`	pairwise cast, (un)box, pad/truncate
`dropArguments`	ignore (*N* consecutive)
`insertArguments`	pre-apply (*N* consecutive)
`permuteArguments`	reorder (also, drop and/or duplicate)
`collectArguments`	collect *N* trailing (enter varargs)
`spreadArguments`	spread *N* trailing (exit varargs)
`filterArguments`	unary compose: *h(f(x), g(y)…, u…)*
`foldArguments`	adjoin: *h(f(x, y…), x, y…, u…)*

Figure 5. Summary of argument adaptation combinators.

The last two, "filter" and "fold", compose two or more method handles in useful patterns. They are general mathematical function compositions but are "adapters" only when regarded as such.

4.4 Bound Method Handles

If arguments can be dropped, they can also be inserted; this is done by so-called *bound method handles*. A bound method handle *b* embodies the partial application of an argument value *k* to a target method handle *h*.

For example, if *h* is a direct handle to a virtual method *m*, it can be bound to a precomputed receiver *k*, producing a wrapper *b* that retains both *k* and *m*. Optimization may pre-compute virtual or interface dispatch logic. Every invocation of *b* accepts the non-receiver arguments $x_{(*)}$, stacks *k* before the other arguments, and executes $k.m(x_{(*)})$.

The bound handle *b* is immutable and opaque. The values *k* and *h* cannot be queried or changed. Whether *b* retains a reference to the reference *h* is an implementation decision.

For example, the handle `println` mentioned above can be bound to the current value of `System.out` as shown in Figure 6. The resulting bound handle will accept a single `String` argument, and print it to the bound output stream.

```
int pos = 0;  // receiver in leading position
MethodHandle println2out = MethodHandles
   .insertArguments(println, pos, System.out);
println2out.<void>invokeExact("hello, world");
```

Figure 6. Creating and using a bound method handle.

Bound method handles support generalization (h) in section 2.1 above, and can be used to implement curried functions and closures. With closures, care must be taken to properly display mutable variables to the target method. In this example, if the value of `System.out` is changed, the bound method handle cannot track this change. Curried functions are a simpler story: The bound method handle represents the intermediate state where the function has received its first argument.

[5] Some roads were not taken: In early draft specifications for JSR 292, the method name was simply `invoke`, but that name is reserved in case of a future need for strongly-typed invocation from Java. We also considered defining a new "`invokehandle`" bytecode to express this operation, but there are no differences from `invokevirtual` significant enough to merit a new bytecode.

Actually, a bound method handle can bind any argument, of any type, in any position, not just a leading reference argument. Cascading method handles can bind multiple arguments.

A bound method handle flexibly combines code and data into a single reference. With this building block, many function-oriented design patterns become more practical in the JVM.

4.5 Java Method Handles

One special sort of method handle mixes object-oriented features into function pointers. A *Java method handle* is created simply by subclassing the abstract type `java.dyn.JavaMethodHandle` and adding any desired fields, behaviors, and API elements. A Java method handle *h* is always created with a target method handle *m*. It behaves (i.e., responds to `invokeExact`) like a bound method handle *b* for which the target is *m* and the bound receiver argument is *h* itself. It differs from such a *b* only in that *h* is its own bound argument. The constructor for `JavaMethodHandle` takes a specification, either a local name or a method handle, of the target *m*.

The result of this self-recursion maneuver is twofold: *h* is a method handle, and *h* is an object with state, behavior, and APIs. One thing *h* cannot do is inherit from another superclass like `Number` or `AbstractList`, but it can implement interfaces. Figure 7 is an example of a method handle with an extra object-like API.

```
abstract class AbstractSettableMethodHandle
    extends JavaMethodHandle {
  private final MethodHandle setter;
  public MethodHandle setter()
    { return setter; }
  public AbstractSettableMethodHandle(
      String get, String set) {
    super(get);  // self is the getter
    MethodType getType = this.type();
    MethodType setType = …;
    this.setter = MethodHandles.publicLookup()
       .bind(this, set, setType);
  }
}
```

Figure 7. Object-like method handle with a "setter" API.

As a matter of programming convenience, Java method handles also allow programmers to use inner class notation when working with method handles. Figure 8 is an example of a curried function `greet` implemented using an anonymous class.

```
MethodHandle greet(final String greeting) {
  return new JavaMethodHandle("run") {
      void run(String greetee) {
        String s = greeting+", "+greetee;
        System.out.println(s);
    } };
}
...
greet("hello").<void>invokeExact("world");
// prints "hello, world"
```

Figure 8. Anonymous class specifying a method handle.

Both of these examples exhibit hybrid object/functional programming patterns. Although the second example may be a mere creature comfort for Java programmers, the first example shows that Java method handles will be needed if method handles are to be first-class values in hybrid languages like Scala [Odersky04] or OCAML [Clerc09].

4.6 Method Handles and Control Flow

Further method handle factories and combinators supply data structure and control flow operations in method handle form. These support field and array element access, conditionals, and exception processing.

The most important method in this last category builds a standard "if-then-else" control structure:

```
∀S λ(x,y,z). λa:S. if (x a) then y a else z a
```

We call it `MethodHandles.guardWithTest`. It is polymorphic across all signatures. It is designed to support type dispatch logic where an expected condition is tested by a predicate which guards a fast path; a slow path (the third argument) is taken as a fallback.

As we will see in section 5 below, `guardWithTest` is enough to implement many dispatch methods such as inline caches. This supports generalization (l) in section 2.1 above.

Moreover, by combining `foldArguments` with a direct method handle to the method `MethodHandle.invoke` itself, multi-way branch combinators can be created such as this one:

```
λ(S,disp). λ(x,*a:S). disp(x).invokeExact(x,*a)
```

Here, `disp` is a dispatch function which looks at a receiver argument `x` and returns a method handle to invoke on `x` and the rest of the arguments. The parameter `S` is a `MethodType` required to select the right overloading of `MethodHandle.invokeExact`.

4.7 Security Considerations

The JVM has important security features, notably link-time access checking, call-chain permission checks, and class loader constraints. JSR 292 features add nothing new to this mix, except for creator-based access checking of direct method handles, as described in section 4.1 above. Direct method handles add no loopholes; with respect to access checking they are equivalent in power to inner classes.

The setup and linking of a dynamic call site is secure, because only the class containing the call site, or a privileged third party, may specify a bootstrap method. The target of a call site may be set without any access checking, but the call site itself is a reference created by the bootstrap method, and shared only with the JVM. As long as the call site reference is shared only with trusted allies, no attacker can read or write its target.

In the JVM, a privileged operation is allowed or denied based on the callers of the operation. This is done by examining the call chain, either directly by walking the stack or by looking at stack summaries. Thus, changing the call chain can perturb security checks. A call to a direct method handle with no adaptation produces the same set of stack frames as a normal call. However, a call with signature adaptation or filtering can require extra adapter frames within the JVM. These extra frames must neither increase nor decrease permissions for checks based on call chains.

A method handle call performs signature checking against the handle's `MethodType`, which is a reified signature containing `Class` references, not a symbolic value containing merely their names. For this reason, class loader constraints do not need to be checked across method calls. If a constraint would be violated by a normal method call, a `WrongMethodTypeException` results from the analogous method handle call.

The JSR 292 draft API contains no standard combinators to execute `AccessController.doPrivileged` or similar methods. They can be easily defined by user code.

5. Case Study: Inline Caches and `invokedynamic`

An *inline cache* is a call site which keeps track of the locally observed type of a message receiver. It speculatively optimizes the case of a repeated type by directly jumping to a pre-dispatched method. This technique was created for Smalltalk a quarter century ago [Deutsch84]. In this usage, the term *inline* means *local*.

A message which potentially dispatches to different methods for varying receiver types is called polymorphic. By contrast, when a receiver type appears to be actually constant at a call site, that call site is called *monomorphic*. The key insight behind the inline caching optimization is that most call sites are monomorphic. This is true even though many call sites use messages which are polymorphic when considered across the whole program.

When they are tracked locally enough, operand type statistics are highly correlated and biased. To exploit this regularity, a number of interrelated tactics are used in modern systems: [6]

1. Operand types are collected at message sends and other opportune points, and are captured in inline caches or summarized in a *type profile*. This information is local to each instruction.
2. Instructions are split by trace splitting or inlining, producing refined caches or profiles. The splits can be chosen so that the increased locality across different traces of the same original code is likely to exhibit distinct type statistics.
3. Operand types are predicted, based on statistics, *a priori* expectations, local inference, global analysis, or all of the above.
4. Speculative optimizations are placed on a *fast path* protected by a *guard predicate* testing a speculated condition. Each fast path is joined by a *slow path* which handles guard failures.
5. Slow paths may execute more general code or branch to another execution mode, such as an interpreter. They may also trigger a discard of the current optimized code, queue it for future re-optimization, or patch the code[7] of the fast path.
6. If the slow path does not merge with the fast path's successors, the fast path guard dominates them, allowing speculative type predictions to flow forward in the control flow graph.
7. A runtime dispatch can be speculatively resolved to a method, which then may be called directly, after a simple type guard. The guard is usually a pointer comparison with a predictable branch. This tactic strength-reduces complex dispatch logic such as a virtual table lookup or method table search.
8. Speculatively dispatched methods can be inlined, and the inlined code can be customized to all relevant locally predicted operand types. Sometimes an out-of-line call is preferable.
9. If a dynamic compiler runs after type information is collected, it uses this type information to guide the previous steps.

A basic combination of these techniques is a patchable call site which is guarded by a type test, whose fast path directly invokes a predicted method, and whose slow path updates the state of the call site to adapt to a previously unexpected type.

[6] Beyond Smalltalk 80, many combinations of these techniques were first explored in the Self project [Chambers91], and they continue to be used in present-day high performance JVMs. Smalltalk and Self were strong influences on their design. The product JVMs of Sun and IBM started out as Smalltalk engines.

[7] In a multiprocessor system, patching code is a difficult task. The various techniques for patching safely are worth recounting but are beyond the scope of this paper.

The initial state of such a call site is unbiased to any particular type, and may be called neutral or unlinked. When the optimization works, the site stays monomorphic after the first use, specialized to a particular type. If too many types appear, the call site is patched to a *megamorphic* state in which all types can be handled robustly.

5.1 Inline Caches for JVMs

Specifically, consider an inline cache for `Number.intValue`, in the typed Java expression `y=x.intValue()`. Suppose that after some time it has been specialized to `Integer`. Then the call site logic may be described by the pseudocode in Figure 9.

```
Number x = …;
int y;
if (!(x instanceof Integer))
  /*S*/ y = x.intValue(); // invokevirtual
else
  /*F*/ y = ((Integer)x).value;
```

Figure 9. Pseudocode for inline cache logic of `invokevirtual`.

The fast path, marked *F*, is the inlined `intValue` method for `Integer`, which loads the `value` field from the box class. The slow path, marked *S*, is taken when `x` appears with an unexpected type, such as `BigInteger` or `Double`. It may simply be a multi-way call through the virtual table of `Number`. The slow path will also cause the JVM to consider modifying this call site.

What does this technique look like with `invokedynamic`? The essential elements are call site modification via `setTarget`, the `guardWithTest` combinator to protect the fast path, a subclass of `CallSite` to manage cache state and policy, and a method handle to manage the slow path, all bound to the call site.

To be specific, let us assume an original untyped expression `y=x+1`. Dispensing with configuration and bootstrapping details, suppose the site has already been through the following events:

- The expression has been compiled to bytecode as a dynamic call site, with a signature of ⟨`Object`, `int`, `Object`⟩.
- The bootstrap method has reified it as an `ICCallSite`, a class defined by some language runtime.
- After many calls, `x` has always been an `Integer`.
- The call site is eventually relinked to a `guardWithTest` which checks this assumption and calls a method specialized to the primitive type `int`.

In the steady state, the execution of the call site proceeds as if through the pseudocode in Figure 10.

```
Object x = …, y;  // y = x+1
static final ICCallSite site = …;
static final MethodHandle logic = …;
if (site.target != logic)
  /*L*/ y = site.target.invokeExact(x,1);
else if (!(x instanceof Integer))
  /*S*/ y = site.cacheAndCall(x,1);
else
  /*F*/ y = Runtime.add((int)(Integer)x,1);
```

Figure 10. Pseudocode for inline cache, at a dynamic call site.

There is no particular class hierarchy here, just the concrete type `Integer` and a static method in a fictitious class `Runtime`.

Note that the right-hand operand, the constant `1`, is represented as the primitive type `int` in the signature of the call site, instead of boxing it. Specialized argument types at dynamic call sites defer boxing or casting to the language logic which resolves the call site, making the bytecode simpler and allowing more flexibility during linking. In this case, we assume there is a routine `Runtime.add` which is specialized to the `int` primitive type. It might contain only an `iadd` instruction without overflow checking and returning an `int`, or it might have an overflow path which returns a `BigInteger` or `Double` through an `Object` return type.

The linkage path, marked *L*, is taken only if the call site is relinked to a new target. This could happen after `x` begins to take on unexpected types. Compiled code for this path may have an explicit check, or else there may be a barrier in the `CallSite.setTarget` method which triggers patching of the code. If the call site turns out to be megamutable, only the *L* path is likely to be compiled.

The slow path, marked *S*, is the fallback side of a `guardWithTest` combinator, taken if the optimistic type test fails. Since the slow path needs a reference to the reified call site, the slow path will be a bound method handle pre-applied to the site. The fictitious `cacheAndCall` method of `ICCallSite` will inspect the type of `x` and dispatch, in a language-specific way, to a suitable target method, or else perform language-specific error processing. The `cacheAndCall` method also keeps track of the observed types of `x` and manages the call site target accordingly. It is this path which earlier recognized that the type of `x` is always `Integer` and put the logic of Figure 10 into the site as a combined method handle.

Finally, the fast path, marked *F*, calls a target method specialized for integer operands. The target method takes primitive `int` arguments. The unboxing of `x`, which appears as a cast in the pseudocode, is supplied by an adapter method handle wrapped around `Runtime.add`.

5.2 Optimizing an Inline Cache

In its classic form, an inline cache is not easily optimized, because it takes the fixed form of a patchable out-of-line call, associated with a patchable test. However, because code shapes like the examples above can be understood by an optimizing compiler, much more can be done.

If the compiler knows enough type history to speculate that a virtual call site will stay monomorphic, it can inline the target method. Likewise, if an optimizing compiler can speculate that a dynamic call site is not megamutable, it can inline the target method handle graph.

Inlining through method handle calls is practical because method handle graph structure is immutable and scrutable; it can be walked by the compiler. The graph can be inlined whenever the root method handle can be constant-folded.

In the examples in Figures 9 and 10 above, it is likely that an optimizing dynamic compiler will succeed in inlining and optimizing from the `invoke` instruction all the way into the target method. This is feasible because the fast paths are scrutable and are gated by slowly changing predicates. In particular, the `invokevirtual` of `Number.intValue` may well inline up to the `getfield` instruction which fetches the `Integer.value` field of `x`. The `invokedynamic` instruction may well inline all the way to the `iadd` instruction at the heart of `Runtime.add`.

Inlining is a most important optimization because it tends to produce many additional optimization opportunities as the inlined code is integrated into the code around the call site. Type and value inferences can spread to more use points. If loops and other hot code can be fully inlined into all call sites, then more sophisticated escape analyses and loop transformations can apply. As the "optimization horizon" widens, more and more techniques can synergistically apply. Compilation can be viewed as a process which starts with a narrow selection of source code and gradually enlarges the optimization horizon until a large body of source code can be transformed collectively into efficient machine code.

Many of the optimizations described in this section are speculative. By inlining through the mutable target of a dynamic call site, the dynamic compiler gives language logic a "hook" for making language-specific speculations, and embedding the resulting guards and fast paths into optimized code.

The assumed stability of dynamic call site targets, plus the scrutability of method handle graphs, introduces method handles as a second intermediate language in the JVM, on a par with bytecode. Each language can incorporate expressions from the other by direct reference. Bytecode incorporates method handles via `invokedynamic` instructions, while a direct method handle incorporates the bytecode of the Java method to which it refers.

Since call site splitting is an important transformation, the JSR 292 draft specification also allows the JVM to split dynamic call sites. This has the perhaps surprising effect of invoking the bootstrap method to reify new copies of previously reified call sites, addressing pain point 12 in section 2.2 above.[8]

5.3 Many More Variations

The above example of `invokedynamic` can be varied by changing the language logic embodied in the reified call site and/or the bootstrap method. For example, decision trees for polymorphic inline caches can be composed from multiple `guardWithTest` combinators. Multi-way branch combinators be used instead of guards. The bootstrap method can hand the JVM a plain `CallSite`, or it can use a language-specific subclass equipped with fields for counters, type summaries, version numbers, etc.[9]

Our example extends readily to multiple dispatch. Instead of the expression `x+1` with its known right-hand type of `int`, the expression `x+w` can be compiled by replacing `int` values with `Object` or `Integer` values, and guarding the fast path by testing both operands for type `Integer`. Languages with generic arithmetic have a powerful new tool for optimizing on the JVM.

Multiple dispatch may also be used to emulate overloaded methods for Java, as is done by the "POJO Linker" of the Dynalang Project [Szegedi09]. Simple linking to Java methods can be accomplished in the bootstrap method by setting the target to a direct method handle, and forgetting about the reified call site.

Mixed-mode JVMs like HotSpot delay dynamic compilation of methods, causing a different sort of state change at call sites as executable representations are updated. This pattern becomes available to language logic via mutable dynamic call sites. A scripting language runtime may organize its program representation in several formats, starting with text fragments, moving to abstract syntax trees (ASTs), and finally optimizing to bytecode. An AST is equipped with its own interpreter. If the bytecode is a foreign instruction set with an associated interpreter, bytecode fragments would be bound to that interpreter. These varying representations must at some point be managed under a common invocable type, and method handles fill this role well. In particu-

[8] The author is indebted to Rémi Forax and Neal Gafter for insisting on the need for JVM-directed splitting of dynamic call sites.

[9] It is possible to retract the use of a specialized `CallSite` subclass by unlinking the call completely. This will force a new trip through the bootstrap method, to make a new `CallSite`.

lar, a text fragment, AST, or foreign bytecode set can be bound, as data, to an evaluator (a Java method) in a bound method handle.

For example, each JRuby AST node has an `interpret` method which executes it. Binding such a node to its `interpret` method will create a bound method handle that behaves like an entry point into the JRuby AST interpreter, starting at that node. If JRuby decides to compile the AST, it might replace uses of the method handle by a direct method handle to a bytecoded method.

6. Early Adopter Experience

By using custom-written simulator objects instead of method handles and static final variables as call site roots, it is possible to gain some of the optimizations that `invokedynamic` provides, without using JSR 292. The JRuby engineers, consulting with the HotSpot compiler team, spent many months optimizing their inline caching call paths to map better to HotSpot's native inline caching optimization. Using simulator objects, they implemented language-specific inline caches and delayed compilation from AST to bytecode, techniques which we described above in sections 5.1 and 5.3, respectively. The result was a Ruby implementation which was about double the performance of the standard C-based version [Cangiano08]. More recently, they have adopted JSR 292 functionality, and found that their implementation has become simpler, with no loss of performance [Nutter08]. This is because JSR 292 provides a way to express their intentions to the JVM more directly. The performance will continue to improve as the JVM does a better job optimizing `invokedynamic`.

Another relevant use of simulator objects is the JSR 292 Backport [Forax09]. This project emulates `invokedynamic` in older JVMs by editing bytecode as it is loaded, expanding `invokedynamic` and method handle invocation points into simulation code. To gain the inlining required for best performance, hot method handle graphs are detected and converted to bytecode. The bytecode is generated on the fly by walking the method handle graphs, and then handed to the JVM for optimization. After loading and warm-up, the performance is similar to the current version of JSR 292. We expect that performance will be best in the native version, because the inlining decisions will be carried out in one place, inside the dynamic compiler.

7. Comparative Intermediate Languages

The beginning of this paper pointed out that JVM bytecode is designed to be compact, executable, translatable, portable, verifiable, modular, and easy to read and write. These are all characteristics of a good intermediate language or representation.[10]

An intermediate language is useful because it provides an interface between a wide variety of producers and consumers. Producers of bytecode include source compilers, assemblers, and dynamic code generators. The consumers are the interpreters and dynamic compilers of JVMs, plus various module managers, debuggers, and programming environments. Some producers are also consumers, such as annotation processors, aspect engines, and obfuscators. The power of an intermediate language comes from the variety of its *M* producers and *N* consumers, and also from the rich interaction of the $M{\times}N$ combinations between producers and consumers.

[10] Exactly when a representation becomes a language is a question we will not attempt here, except to observe that languages seem to be less context-dependent and more relocatable than other data structures. This may be more a difference of degree than of kind.

To evaluate the needs of languages on the JVM, we must be aware of the strengths and weaknesses in existing implementations, especially of dynamic languages. The first executable form of such a language is usually an AST, which is walked by an interpreter. If actual JVM method invocation is done at a centralized place in the interpreter, the type history of every source-level call is collected by the JVM, in a giant reverse split, into a single megamorphic type profile. This is pain point 4 from section 2.2.

Moreover, special semantics such as JavaScript support for XML objects [Flanagan06] are often coded, as in Rhino [Mozilla09], by hard-coding extra type tests in central interpreter routines, whose increased bulk becomes harder to optimize. A more modular solution is to use a metaobject protocol [Kiczales91, Szegedi09] to control method linkage and dispatch.

A final problem with an AST is its richness; usually it is printable, debuggable, editable, etc. Because one can do more with an AST than simply executing it, it is hard to isolate the executable slice of it and optimize down to machine code that *only* executes.

Most projects probably stop with an AST, perhaps adding inline caches. Some compile to JVM bytecode and rely on the JVM to manage optimization; Clojure is successful at this because its simulation objects are low-level and hence scrutable to the JVM. Others like Rhino compile the interpreter calls made at the leaves of the AST into a sequence of calls to the interpreter runtime. This is only modestly faster (often about 2×) than AST walking.

Interestingly, Microsoft's DLR uses an AST [Hugunin07] for specifying dynamic behaviors. This AST is better suited for the purpose than user-designed ASTs because it (a) is matched to language neutral bytecode operations and (b) is consumed directly by the dynamic compiler. Our method handle graphs play a similar role. In comparison with DLR AST, they are opaque and somewhat coarse-grained, since they cannot specify individual variable references, or directly specify instructions like `iadd`. On the other hand method handles are directly invocable.

An *unused* tactic is bytecode generation with call site editing, even though this is commonly done by JVMs at the machine code level. The bytecode format is rigid. To change a single method call after it has been linked, a new set of bytecode must be assembled, loaded, and somehow replaced over the old version. The JVM must recheck and re-optimize not only the changed method call, but all the surrounding code. Such *hot swapping* is possible, but making it complete, correct, and efficient is still a research topic [Subram09, Wuerth09]. Hot swapping is sometimes used to implement aspect-oriented programming [Nicoara08], but using it for dynamic languages seems a dim hope.

There is one more objection to dynamic bytecode generation on memory limited mobile devices. Such systems often need to know code sizes early; some disallow dynamic code generation.

The awkwardness of incrementally generating or transforming bytecode is a consequence of its compactness, and its designed-in similarity to machine code. By contrast, the pointer-rich structure of method handles makes them easy to patch together.

Method handles are not always the best choice. Their opaqueness makes them difficult to transform or to transport outside of the JVM that created them. They are unsuitable for representing contextual or frame-relative operations, such as local variable definition. If not compiled, they are less compact and slower to execute than bytecode, especially for complex code.

There seems to be a good impedance match between bytecode and method handles. The dynamic overhead of shifting between representations is low, and there is parity between them in expressive power. Although coding with method handles involves a certain amount of boilerplate, our experience of programming with the dual representations has been rewarding so far.

8. Conclusion

The JSR 292 design folds inline caches and method handles into the JVM's intermediate language. The `invokedynamic` instruction is a hinge-point between two complementary kinds of intermediate language: bytecode and combinators.

Bytecode is still the most efficient and compact representation for computations which are statically fixed. Conversely, method handle graphs are the most efficient way to glue together computations on the fly. Both representations are directly executable and compilable. Both have the virtues of easy construction, modularity, type safety, security, and standardization.

The two languages can be woven together. Bytecode contains dynamically editable holes at `invokedynamic` instructions. Method handle graphs, though not statically constructed, are woven into those bytecode holes. The combination can be inlined, optimized, and compiled down to efficient machine code.

Picking the right language for the job is an old theme, which is now gaining acceptance as compelling new languages begin to run on the JVM. We hope to empower implementors by giving them *their* choice of the right intermediate language for each job.

Acknowledgments

This work was funded by Sun Microsystems and guided by the Java Community Process JSR 292 Expert Group. Numerous colleagues have contributed to the design, especially through the open-source Da Vinci Machine Project. Charles Nutter and the JRuby implementation team have been vigorous customers of this functionality from the beginning. In addition to the anonymous referees, Paul Hohensee, Alex Buckley, and Dean Long gave indispensable early comments on this paper.

References

[Cangiano08] Antonio Cangiano. The Great Ruby Shootout. Dec. 2008. URL: http://antoniocangiano.com/2008/12/09/the-great-ruby-shootout-december-2008

[Chambers91] Craig Chambers, David Ungar, Elgin Lee. An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Lisp and Symbolic Computation* 4, 3 (1991), 243–281. Also in OOPSLA '89 Conference Proceedings.

[Clerc09] Xavier Clerc. Ocaml-java project (ca. 2009). URL: http://ocamljava.x9c.fr

[Deutsch84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. *POPL* (1984) 297-302.

[DaVinci09] Da Vinci Machine project (ca. 2009). URL: http://openjdk.java.net/projects/mlvm/

[Flanagan06] David Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, Inc., Sebastopol, CA (2006), 21.8, p. 527.

[Forax09] Rémi Forax. JSR 292 backport – First release. July, 2009. URL: http://weblogs.java.net/blog/2009/07/01/jsr292-backport-first-release

[Goldberg83] Adele Goldberg, David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley (1983).

[Gosling95] James Gosling. Java intermediate bytecodes: ACM SIGPLAN workshop on intermediate representations (IR '95), San Francisco (1995), 111-118.

[Hickey09] Rich Hickey et al. Clojure project (ca. 2009). URL: http://clojure.org/

[Hugunin07] Jim Hugunin. DLR Trees (Part 1). Microsoft, May 2007. URL: http://blogs.msdn.com/hugunin/archive/2007/05/15/dlr-trees-part-1.aspx

[Hugunin09] Jim Hugunin et al. Jython project (ca. 2009). URL: http://www.jython.org

[Kiczales91] Gregor Kiczales, Jim des Rivieres and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press (1991).

[Lindholm99] Tim Lindholm, Frank Yellin. *Java Virtual Machine Specification*, Addison-Wesley, Reading, MA (1999). URL: http://java.sun.com/docs/books/jvms/

[Lorimer09] R.J. Lorimer. Distilling JRuby: Method Dispatching 101. September, 2009. URL: http://www.realjenius.com/2009/09/16/distilling-jruby-method-dispatching-101/

[Meijer00] Erik Meijer, John Gough. Technical Overview of the Common Language Runtime. Microsoft (2000). URL: http://research.microsoft.com/~emeijer/Papers/CLR.pdf

[Mozilla09] Mozilla Corp. Rhino: JavaScript for Java. URL: http://www.mozilla.org/rhino/.

[Musch08] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple Dispatch in Practice. In *OOPSLA*, pages 563–582, Nashville, TN, USA (2008).

[Nicoara08] Angela Nicoara, Gustavo Alonso and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running java programs. In SIGOPS/EuroSys (Glasgow, Scotland UK, 2008).

[Nutter08] Charles Nutter. A First Taste of InvokeDynamic. Sept. 2008. URL: http://blog.headius.com/2008/09/first-taste-of-invokedynamic.html

[Nutter09] Charles Nutter et al. JRuby project (ca. 2009). URL: http://kenai.com/projects/jruby/

[Odersky04] Martin Odersky et al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland (2004).

[Ohloh09] Ohloh Company. Compare Languages Tool. (Retrieved 10/2009.) URL: http://www.ohloh.net/languages/compare

[Öhrström09] Fredrik Öhrström. The JSR292 endgame. Oracle, May 11, 2009. URL: http://blogs.oracle.com/ohrstrom/2009/05/the_jsr292_endgame.html

[Rose04] John Rose. Tuples in the VM. August, 2004. URL: http://blogs.sun.com/jrose/entry/tuples_in_the_vm

[Rose08] John Rose. Symbolic Freedom in the VM. January 23, 2008. URL: http://blogs.sun.com/jrose/entry/symbolic_freedom_in_the_vm

[Schwaig09] Arnold Schwaighofer. *Tail Call Optimization in the Java HotSpot VM*. Master's thesis, Johannes Kepler University Linz, Austria (2009).

[Stadler09] Lukas Stadler et al. Lazy Continuations for Java Virtual Machines. In *Principles and Practice of Programming in Java*, Calgary, Alberta, Canada (2009).

[Steele96] James Gosling, Bill Joy, Guy L. Steele, *The Java Language Specification*, Addison-Wesley (1996). URL: http://java.sun.com/docs/books/jls/

[Subram09] Suriya Subramanian, Michael Hicks and Kathryn S. McKinley. Dynamic software updates: a VM-centric approach. In *PLDI*, Dublin, Ireland (2009).

[Szegedi09] Attila Szegedi. Metaobject Protocol Meets Invokedynamic. JVM Language Summit, Santa Clara (2009). URL: http://wiki.jvmlangsummit.com/MOP_and_Invokedynamic Project URL: http://dynalang.sourceforge.net/

[TIOBE09] TIOBE Software BV. TIOBE Programming Community Index, Long Term Trends. (Retrieved 10/2009.) URL: http://www.tiobe.com/index.php/tiobe_index

[Tolksdorf09] Robert Tolksdorf. Programming languages for the Java Virtual Machine JVM. URL: http://www.is-research.de/info/vmlanguages/

[Wuerth09] Thomas Wuerthinger. Dynamic Code Evolution for the Java HotSpot Virtual Machine. April 1, 2009. URL: http://wikis.sun.com/download/attachments/172493511/wuerthinger-hotswap-20090401.pdf