Using invoke dynamic to teach the JVM a new language

Jonathan Worthington

Things I work on

EDUMENT
Development and Mentorship

Perl

Writing and teaching courses, mostly about software architecture, TDD and C#

Lead developer and architect of the Rakudo Perl 6 compiler; focus on OO, type system, etc.

Various bits of mentoring and consulting

Various other contributions (native calling, debugger, ...)

Bringing Perl 6 to the JVM

Over the last year, I've been working on bringing Perl 6 to the Java Virtual Machine

Many languages have a JVM port these days

In JDK7, the invokedynamic instruction was added

Its initial aim was to help those writing compilers for languages of a more dynamic nature. However, it will also be used by Java 8's lambda support.

Perl 6 in 2 minutes

No, not a 2-minute language tutorial

Just enough to give the context in which I've been using invokedynamic

Perl 6 has an executable specification

```
subset Even of Int where { $_ % 2 == 0 };

{
    my Even $x = 2;
    is $x, 2, 'Can assign value to a subset-typed variable';
};

dies_ok { eval('my Even $x = 3') },
    "Can't assign value that violates subset type constraint";
```

With Perl 5, the implementation was the spec

For Perl 6, there is a written specification, though it's fairly informal. The real, formal, specification is the standard grammar and a specification test suite.

Perl 6 is multi-paradigm

```
multi quicksort([]) {
    ()
}

multi quicksort([$pivot, *@rest]) {
    quicksort(@rest.grep(* < $pivot)),
    $pivot,
    quicksort(@rest.grep(* >= $pivot))
}

say join ", ", quicksort([1, -2, 5, 8, -6, 3])
```

Perl 6 is multi-paradigm

```
multi quicksort([]) {
    ()
}

multi quicksort([$pivot, *@rest]) {
    quicksort(@rest.grep(* < $pivot)),
    $pivot,
    quicksort(@rest.grep(* >= $pivot))
}

say join ", ", quicksort([1, -2, 5, 8, -6, 3])
```

Object oriented: everything can be treated as an object, if you wish to work that way...

Perl 6 is multi-paradigm

```
multi quicksort([]) {
    ()
}

multi quicksort([$pivot, *@rest]) {
    quicksort(@rest.grep(* < $pivot)),
    $pivot,
    quicksort(@rest.grep(* >= $pivot))
}

say join ", ", quicksort([1, -2, 5, 8, -6, 3])
```

Procedural: just because we've classes, roles, and so forth doesn't mean a function isn't sometimes best!

Perl 6 is multi-paradigm

```
multi quicksort([]) {
    ()
}

multi quicksort([$pivot, *@rest]) {
    quicksort(@rest.grep(* < $pivot)),
    $pivot,
    quicksort(@rest.grep(* >= $pivot))
}

say join ", ", quicksort([1, -2, 5, 8, -6, 3])
```

Higher order programming: various ways to write closures, including the * auto-closure syntax

Perl 6 is multi-paradigm

```
multi quicksort([]) {
    ()
}

multi quicksort([$pivot, *@rest]) {
    quicksort(@rest.grep(* < $pivot)),
    $pivot,
    quicksort(@rest.grep(* >= $pivot))
}

say join ", ", quicksort([1, -2, 5, 8, -6, 3])
```

Pattern matching as found in many functional languages - integrated with multiple dispatch

Perl 6 is gradually typed

```
my num $distance = get-distance();
my num $time     = get-time();
say "Speed = { $distance / $time }";
```

You don't have to write any type annotations

But if you do, the compiler will make use of them, both for optimization and error reporting

Perl 6 is gradually typed

```
my num $distance = get-distance();
my num $time     = get-time();
say "Speed = { $distance / $time }";
```

You don't have to write any type annotations

But if you do, the compiler will make use of them, both for optimization and error reporting

For example, knowing both operands are native floating point numbers, we can compile this division directly to the VM's floating point division op.

Perl 6 lexically scopes language mutation

```
{
    sub postfix:<!>($n) { [*] 1..$n }
    say 10!; # Works
}
# Outside that scope, no ! operator again
```

Yes, you can declare new operators. Here we add a factorial operator, implemented using reduction on the multiplication operator.

But it only exists within the scope where it is declared. Same for language tweaks you import.

Careful dynamic/static trade-off

Nearly all operators are multi-dispatch sub calls

but

We know the candidate list in a given scope

You can run code during the parse, and do whatever meta-programming you like

but

Short of a declaration otherwise, we assume you're done meddling by the time we reach the optimizer

Most dynamic code is eventually static

We often divide languages up into dynamic and static, but program behaviour is mostly static

Sure, if we don't know the type of `$order`, then...

```
$order.change_shipping_category(Urgent);
```

...could be going pretty much anywhere

But in reality, even code that is potentially highly polymorphic from the compile-time view is often monomorphic at runtime

Method calls: what we know in Java

When we compile a method call in Java...

```
order.changeShippingCategory(ShippingCategory.Urgent);
```

...then we know a few things to help us:

Is `order` an interface type or a class type?

Does that interface or class have a method with the name `changeShippingCategory`?

If it's a class type, is the method final or not?

In Java, we know how we don't know

If the method is final, we know where we're going

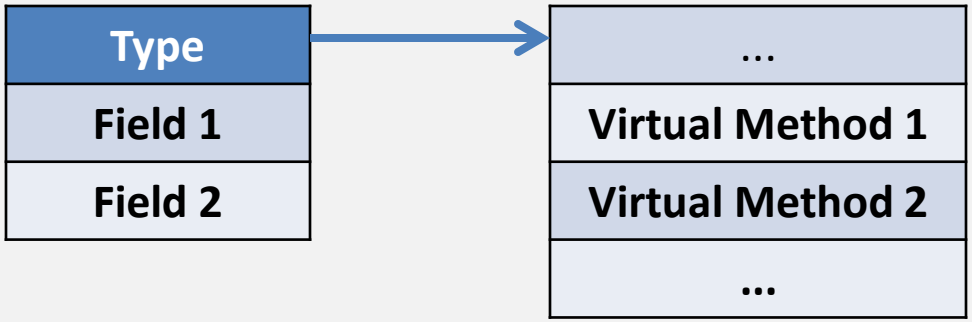
If the method is not final, and on a class, we don't know. But we know how we don't know: we'll need to look in the v-table to find out, at runtime

If the method is on an interface, we also don't know. But again, we know how we don't know: we'll need to look in the interfaces table to find out, at runtime

In Java, method dispatch has known unknowns

Optimize away the unknown!

Naively, a virtual method dispatch can JIT-compile to a lookup into the v-table



But that means we can't inline, which means we can't do escape analysis and other such optimizations

So, the JIT works out ways it can inline, by doing type specialization or speculative optimization

There are more ways of not knowing

Thus, JIT compilers for so-called static languages often already optimize away dynamic behaviour

However, the ways to exploit this have been a poor fit for the unknowns of dynamic languages

Enter invokedynamic

The JVM's `invokevirtual` and `invokeinterface` instructions are ways of "not knowing"

However, they are tied to very particular and inflexible ways of finding out

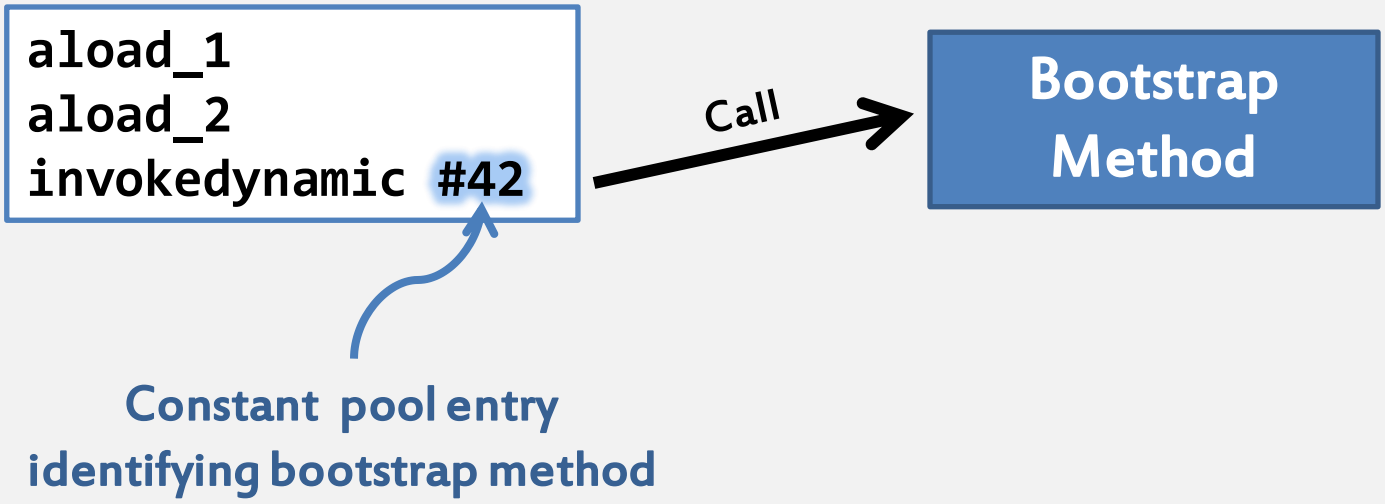
The new `invokedynamic` instruction is also a way of indicating we don't know what to do

However, it does not specify a resolution mechanism. Instead, we can provide the answer!

The basic mechanism (1)

A given usage of invokedynamic specifies a bootstrap method, which decides what to do

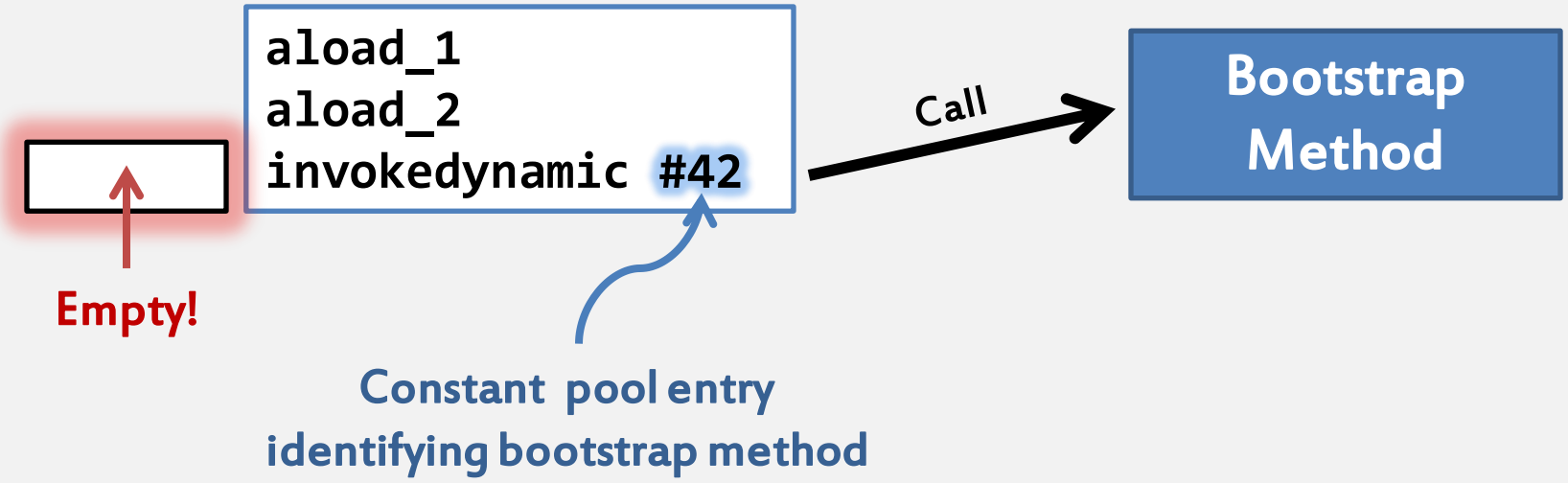
Whenever we encounter an invokedynamic instruction for the first time, the bootstrap is called



The basic mechanism (2)

For each invokedynamic instruction in the bytecode, the JVM can store a reference to a `CallSite` object

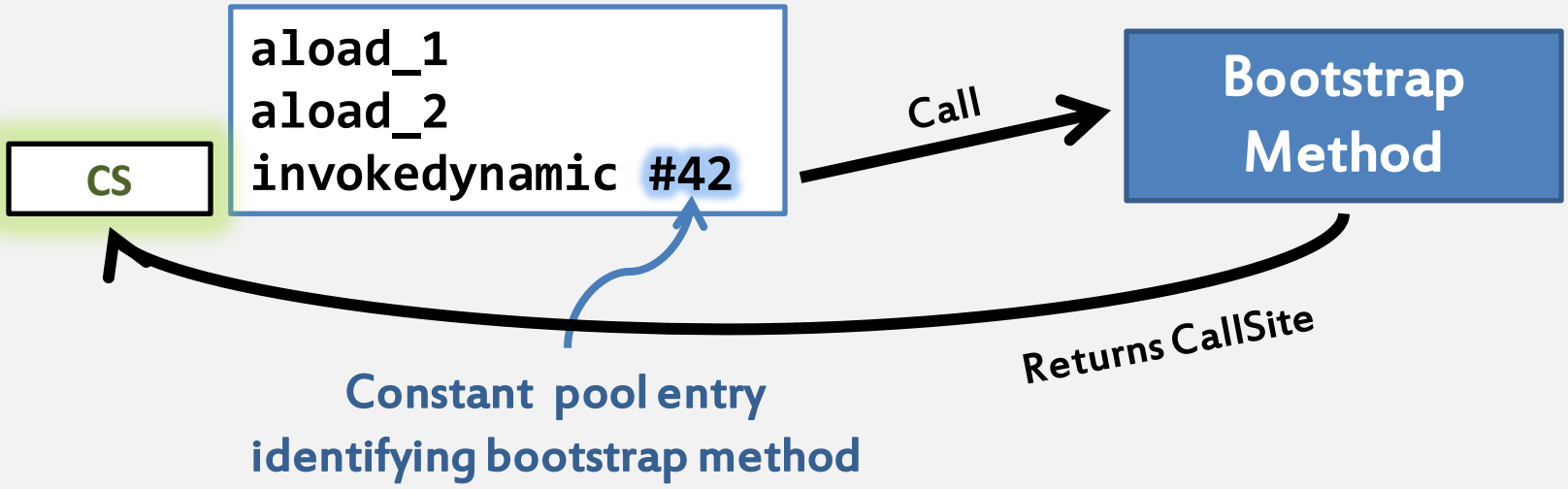
A CallSite specifies a target method reference, which indicates what should be called in the future



The basic mechanism (3)

The bootstrap method creates a `CallSite`. There are various kinds: constant, mutable, volatile.

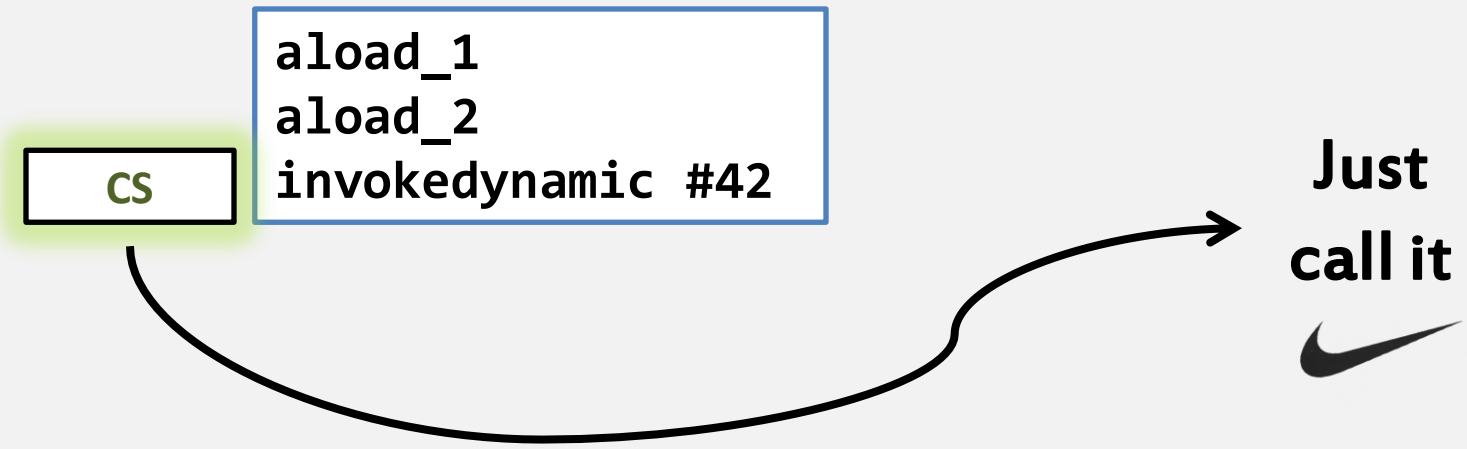
This is then returned from the bootstrap method, and installed in the callsite slot by the JVM



The basic mechanism (4)

The next time we encounter that invokedynamic instruction, we simply call whatever method the callsite's target indicates

May change over time for mutable/volatile sites



Wait, what's a method handle?

A reference to a method, incorporating information about its parameter types and return type

Wait, what's a method handle?

A reference to a method, incorporating information about its parameter types and return type

Well, actually, a reference to something we can invoke, with parameter and return type information

Wait, what's a method handle?

A reference to a method, incorporating information about its parameter types and return type

Well, actually, a reference to something we can invoke, with parameter and return type information

In fact, a reference to a thingy that somehow turns values of certain types into a result of a given type

Wait, what's a method handle?

A reference to a method, incorporating information about its parameter types and return type

Well, actually, a reference to something we can invoke, with parameter and return type information

In fact, a reference to a thingy that somehow turns values of a certain type into a result of a given type

Actually, it's this non-specificity that makes it such a powerful mechanism!

Case studies

To make this a little more concrete, let's consider two cases where I've used invokedynamic while working on the Perl 6 JVM support

Referencing meta-objects

Declarations of classes, attributes and methods result in meta-objects, created at compile time

```
class Meeting {
    has $.start_time;
    has $.duration;

    method end_time() {
        $.start_time + $.duration
    }
}
```

The meta-type used for `class` can be changed in a given lexical scope, however. For example, you may use a module that extends classes with AOP support.

The problem

This, and other features, mean that we need a way for arbitrary objects created during compile time to survive and be referred to at runtime

There can easily be a process boundary here

Can also have cross-references to objects that are persisted in other compilation units

Way, way beyond what we can expect the JVM's constant pool to cope with

Possible solutions

Store all referenced things in comp-unit fields

Can make them final, so JVM knows they won't change. But need to populate fields - used or not - or emit some kind of thunk code to do it. Possible, though it's a little messy.

Store all referenced things in an array

Again, needs setup. Doesn't lead to a huge fields table, however. Problem is how to indicate that elements in an array won't change. Limits how well the JVM can optimize things?

Let's use invokedynamic!

To locate the value, we need a string which is the unique ID of the compilation unit the object was declared and serialized in, and an index

We load these onto the stack. Thus, the bootstrap method needs to install something that will take these two identifiers and resolve them, leaving an object on the stack afterwards.

```
ldc           [String "A3FE2B..."] // Comp unit ID
ldc           [int 42]             // Object index
invokedynamic [Bootstrap.wval]
```

Let's use invokedynamic!

Next up, the bootstrap method

```
public static CallSite wval(Lookup caller, String name,
                            MethodType type) {
    ...
}
```

Let's use invokedynamic!

Next up, the bootstrap method

```
public static CallSite wval(Lookup caller, String name,
                            MethodType type) {
    try {
        ...
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Let's use invokedynamic!

Next up, the bootstrap method

```
public static CallSite wval(Lookup caller, String name,
                            MethodType type) {
    try {
        /* Look up wval resolver method. */
        MethodType resType = MethodType.methodType(P6Object.class,
                MutableCallSite.class, String.class, int.class);
        ...
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Let's use invokedynamic!

Next up, the bootstrap method

```
public static CallSite wval(Lookup caller, String name,
                            MethodType type) {
    try {
        /* Look up wval resolver method. */
        MethodType resType = MethodType.methodType(P6Object.class,
                MutableCallSite.class, String.class, int.class);
        MethodHandle res = caller.findStatic(Bootstrap.class,
                "wvalResolve", resType);
        ...
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Let's use invokedynamic!

Next up, the bootstrap method

```
public static CallSite wval(Lookup caller, String name,
                            MethodType type) {
    try {
        /* Look up wval resolver method. */
        MethodType resType = MethodType.methodType(P6Object.class,
                MutableCallSite.class, String.class, int.class);
        MethodHandle res = caller.findStatic(Bootstrap.class,
                "wvalResolve", resType);

        /* Create mutable callsite, curry resolver with it. */
        MutableCallSite cs = new MutableCallSite(type);
        ...
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Let's use invokedynamic!

Next up, the bootstrap method

```
public static CallSite wval(Lookup caller, String name,
                            MethodType type) {
    try {
        /* Look up wval resolver method. */
        MethodType resType = MethodType.methodType(P6Object.class,
                MutableCallSite.class, String.class, int.class);
        MethodHandle res = caller.findStatic(Bootstrap.class,
                "wvalResolve", resType);

        /* Create mutable callsite, curry resolver with it. */
        MutableCallSite cs = new MutableCallSite(type);
        cs.setTarget(MethodHandles.insertArguments(res, 0, cs));
        return cs;
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Let's use invokedynamic!

Finally, the resolver method

```
public static P6Object wvalResolve(MutableCallSite cs,
                                   String sc, int idx) {
    ...
}
```

Let's use invokedynamic!

Finally, the resolver method

```
public static P6Object wvalResolve(MutableCallSite cs,
                                   String sc, int idx) {
    /* Look up the WVal. */
    P6Object res = GlobalCtx.scs.get(sc).root_objects.get(idx);
    ...
}
```

Let's use invokedynamic!

Finally, the resolver method

```
public static P6Object wvalResolve(MutableCallSite cs,
                                   String sc, int idx) {
    /* Look up the WVal. */
    P6Object res = GlobalCtx.scs.get(sc).root_objects.get(idx);

    /* Update this callsite, so that we never run the lookup again
     * and instead just always use the resolved object. Discards
     * incoming arguments, as they are no longer needed. */
    cs.setTarget(
            MethodHandles.constant(P6Object.class, res));
    ...
}
```

Let's use invokedynamic!

Finally, the resolver method

```
public static P6Object wvalResolve(MutableCallSite cs,
                                   String sc, int idx) {
    /* Look up the WVal. */
    P6Object res = GlobalCtx.scs.get(sc).root_objects.get(idx);

    /* Update this callsite, so that we never run the lookup again
     * and instead just always use the resolved object. Discards
     * incoming arguments, as they are no longer needed. */
    cs.setTarget(MethodHandles.dropArguments(
            MethodHandles.constant(P6Object.class, res),
            0, String.class, int.class));
    ...
}
```

Let's use invokedynamic!

Finally, the resolver method

```
public static P6Object wvalResolve(MutableCallSite cs,
                                   String sc, int idx) {
    /* Look up the WVal. */
    P6Object res = GlobalCtx.scs.get(sc).root_objects.get(idx);

    /* Update this callsite, so that we never run the lookup again
     * and instead just always use the resolved object. Discards
     * incoming arguments, as they are no longer needed. */
    cs.setTarget(MethodHandles.dropArguments(
            MethodHandles.constant(P6Object.class, res),
            0, String.class, int.class));

    /* Hand back the resulting object, for this first call. */
    return res;
}
```

MethodHandle combinators

The `invokedynamic` instruction is just a way of saying, "I don't know"

We supply the answer using method handles

The `MethodHandles` class has many static methods that are either factories or combinators

```
MethodHandles.dropArguments(
    MethodHandles.constant(P6Object.class, res),
    0, String.class, int.class)
```

Combinator Factory

The constant factory

We might imagine a Constant method handle as looking something like this:

```
final class Constant extends MethodHandle {
    private final Object value;
    public Constant(Object value) {
        this.value = value;
    }
    public Object invoke() {
        return value;
    }
}
```

***(From the docs:** Method handles cannot be subclassed by the user. Implementations may (or may not) create internal subclasses of MethodHandle... the method handle class hierarchy (if any) may change from time to time or across implementations from different vendors.)*

The drop combinator

Similarly, we could imagine the drop combinator as looking something like this:

```
final class Dropper extends MethodHandle {
    private final MethodHandle target;
    public Dropper(MethodHandle target) {
        this.target = target;
    }
    public Object invoke(String a, int b, Object more ...)
{
        return target.invokeWith(more);
    }
}
```

How can this be fast?

If the callsite itself points to a Dropper, and we know (or assume, being willing to de-optimize) that it will never change, then JIT can go to work inlining!

Start out with naive code invoking the callsite

```
push "ABCDEF"
push 42
[load callsite]
getfield target    // from callsite
[invoke]
```

How can this be fast?

If the callsite itself points to a Dropper, and we know (or assume, being willing to de-optimize) that it will never change, then JIT can go to work inlining!

Inline Dropper.invoke...

```
push "ABCDEF"
push 42
[load callsite]
getfield target     // from callsite
getfield target     // from Dropper
[invoke]
```

How can this be fast?

If the callsite itself points to a Dropper, and we know (or assume, being willing to de-optimize) that it will never change, then JIT can go to work inlining!

...which eliminates the now-dead pushes:

```
[load callsite]
getfield target    // from callsite
getfield target    // from Dropper
[invoke]
```

How can this be fast?

If the callsite itself points to a Dropper, and we know (or assume, being willing to de-optimize) that it will never change, then JIT can go to work inlining!

Now inline Constant, since target is final:

```
[load callsite]
getfield target     // from callsite
getfield target     // from Dropper
getfield value      // from Constant
```

How can this be fast?

If the callsite itself points to a Dropper, and we know (or assume, being willing to de-optimize) that it will never change, then JIT can go to work inlining!

But if the whole chain is final, we know the value!

```
[load callsite]
getfield target     // from callsite
getfield target     // from Dropper
getfield value      // from Constant
[load object addr]
```

The JIT does what the JIT already did

Reality: I don't actually know for sure that it gets all the way there at present

The point is that even if the JIT compiler is taught very little about invokedynamic and method handles, it can still apply optimizations it already knows in order to make things cheap

Can quite easily imagine it inlining and then lifting guard clauses just out of normal Common Subexpression Elimination, for example

One more example: method calls

We also use invokedynamic for method calls

```
sub enjoy($beer) { $beer.drink() }
```

We don't know the type of $beer. But we don't even know how it dispatches methods! Just imagine...

```
method find_method($obj, $name) {
    if moon_is_blue() {
        die "Cannot call method once in a blue moon";
    }
    else {
        return %!methods_table{$name};
    }
}
```

Plan A: stack up guard clauses

The `guardWithTest` combinator takes three method handles: one that contains a predicate, one to use if it comes out true, and another if it's false

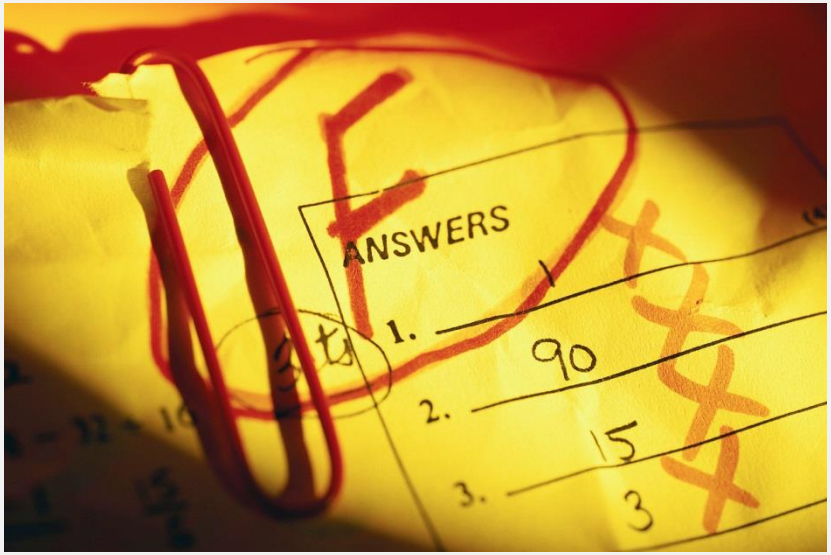
Checking the type of the thing we're calling a method on is cheap compared to a hash lookup

"If the type matches, call the already resolved method, otherwise resolve it the slow way"

JIT can *potentially* inline and coalesce the checks

Plan A: fail!

Actually, it came out a huge amount slower than doing the hash lookup for every single method call



Why? I don't actually know, yet. Didn't have the tuits to investigate and find out.

Plan B: optimize for monomorphism

The bootstrap method installs a first-time-only resolver, that does the method lookup

It then updates the callsite to a second-level resolver that is curried with the type and method

This checks if each future invocant matches the type of the first, and just uses the pre-resolved method handle if so; if not, the normal slow path is used

This one came out faster! ☺

The bootstrap method (simplified)

```
public static CallSite methcall(Lookup caller, String _,
                                MethodType type) {
    try {
        ...
    }
    catch (Exception e) { throw new RuntimeException(e); }
}
```

The bootstrap method (simplified a bit)

```
public static CallSite methcall(Lookup caller, String _,
                                MethodType type) {
    try {
        /* Look up methcall resolver method. */
        MethodType resType = MethodType.methodType(void.class,
                Lookup.class, MutableCallSite.class, String.class,
                Object[].class);
        MethodHandle res = caller.findStatic(Bootstrap.class,
            "methcallResolve", resType);
        ...
    }
    catch (Exception e) { throw new RuntimeException(e); }
}
```

The bootstrap method (simplified)

```
public static CallSite methcall(Lookup caller, String _,
                                MethodType type) {
    try {
        /* Look up methcall resolver method. */
        MethodType resType = MethodType.methodType(void.class,
                Lookup.class, MutableCallSite.class, String.class,
                Object[].class);
        MethodHandle res = caller.findStatic(Bootstrap.class,
            "methcallResolve", resType);

        /* Create a mutable callsite, and curry the resolver. */
        MutableCallSite cs = new MutableCallSite(type);
        cs.setTarget(MethodHandles
            .insertArguments(res, 0, caller, cs)
            .asCollector(Object[].class, type.parameterCount() - 3)
            .asType(type));
        ...
    }
    catch (Exception e) { throw new RuntimeException(e); }
}
```

The bootstrap method (simplified)

```
public static CallSite methcall(Lookup caller, String _,
                                MethodType type) {
    try {
        /* Look up methcall resolver method. */
        MethodType resType = MethodType.methodType(void.class,
                Lookup.class, MutableCallSite.class, String.class,
                Object[].class);
        MethodHandle res = caller.findStatic(Bootstrap.class,
            "methcallResolve", resType);

        /* Create a mutable callsite, and curry the resolver. */
        MutableCallSite cs = new MutableCallSite(type);
        cs.setTarget(MethodHandles
            .insertArguments(res, 0, caller, cs)
            .asCollector(Object[].class, type.parameterCount() - 3)
            .asType(type));
        return cs;
    }
    catch (Exception e) { throw new RuntimeException(e); }
}
```

First level resolver (simplified)

```
public static void methcallResolve(Lookup caller,
        MutableCallSite cs, String name, Object... args) {
    ...
}
```

First level resolver (simplified)

```
public static void methcallResolve(Lookup caller,
        MutableCallSite cs, String name, Object... args) {
    /* Try to resolve method. */
    P6Object invocant = (P6Object)args[0];
    CodeRef  method   = Ops.findmethod(invocant, name);
    STable   type     = invocant.st;
    ...
}
```

First level resolver (simplified)

```
public static void methcallResolve(Lookup caller,
        MutableCallSite cs, String name, Object... args) {
    /* Try to resolve method. */
    P6Object invocant = (P6Object)args[0];
    CodeRef  method   = Ops.findmethod(invocant, name);
    STable   type     = invocant.st;

    /* Update callsite to monomorphic resolver. */
    MethodType resType = MethodType.methodType(void.class,
            String.class, STable.class, CodeRef.class,
            Object[].class);
    MethodHandle res = caller.findStatic(Bootstrap.class,
        "methcallCacheMono", resType);
    ...
}
```

First level resolver (simplified)

```
public static void methcallResolve(Lookup caller,
        MutableCallSite cs, String name, Object... args) {
    /* Try to resolve method. */
    P6Object invocant = (P6Object)args[0];
    CodeRef  method   = Ops.findmethod(invocant, name);
    STable   type     = invocant.st;

    /* Update callsite to monomorphic resolver. */
    MethodType resType = MethodType.methodType(void.class,
            String.class, STable.class, CodeRef.class,
            Object[].class);
    MethodHandle res = caller.findStatic(Bootstrap.class,
        "methcallCacheMono", resType);
    cs.setTarget(MethodHandles.insertArguments(res, 1, type, method)
        .asCollector(Object[].class, cs.type().parameterCount() - 3)
        .asType(cs.type()));
    ...
}
```

First level resolver (simplified)

```
public static void methcallResolve(Lookup caller,
        MutableCallSite cs, String name, Object... args) {
    /* Try to resolve method. */
    P6Object invocant = (P6Object)args[0];
    CodeRef  method   = Ops.findmethod(invocant, name);
    STable   type     = invocant.st;

    /* Update callsite to monomorphic resolver. */
    MethodType resType = MethodType.methodType(void.class,
            String.class, STable.class, CodeRef.class,
            Object[].class);
    MethodHandle res = caller.findStatic(Bootstrap.class,
        "methcallCacheMono", resType);
    cs.setTarget(MethodHandles.insertArguments(res, 1, type, method)
        .asCollector(Object[].class, cs.type().parameterCount() - 3)
        .asType(cs.type()));

    /* Invoke the method. */
    Ops.invoke(method, args);
}
```

Second level resolver (simplified)

```
public static void methcallCacheMono(String name,
        STable assumedST, CodeRef assumedCR, Object... args) {
    ...
}
```

Second level resolver (simplified)

```
public static void methcallCacheMono(String name,
        STable assumedST, CodeRef assumedCR, Object... args) {
    /* See if we match the type of the first call. */
    P6Object invocant = (P6Object)args[0];
    CodeRef method;
    ...
}
```

Second level resolver (simplified)

```
public static void methcallCacheMono(String name,
        STable assumedST, CodeRef assumedCR, Object... args) {
    /* See if we match the type of the first call. */
    P6Object invocant = (P6Object)args[0];
    CodeRef method;
    if (invocant.st == assumedST) {
        /* Same as initial call; know the answer. */
        method = assumedCR;
    }
    ...
}
```

Second level resolver (simplified)

```
public static void methcallCacheMono(String name,
        STable assumedST, CodeRef assumedCR, Object... args) {
    /* See if we match the type of the first call. */
    P6Object invocant = (P6Object)args[0];
    CodeRef method;
    if (invocant.st == assumedST) {
        /* Same as initial call; know the answer. */
        method = assumedCR;
    }
    else {
        /* Polymorphic; need a lookup. */
        method = Ops.findmethod(invocant, name);
    }
    ...
}
```

Second level resolver (simplified)

```
public static void methcallCacheMono(String name,
        STable assumedST, CodeRef assumedCR, Object... args) {
    /* See if we match the type of the first call. */
    P6Object invocant = (P6Object)args[0];
    CodeRef method;
    if (invocant.st == assumedST) {
        /* Same as initial call; know the answer. */
        method = assumedCR;
    }
    else {
        /* Polymorphic; need a lookup. */
        method = Ops.findmethod(invocant, name);
    }

    /* Invoke the method. */
    Ops.invoke(method, args);
}
```

Pushing more knowledge VM-wards

One other important side-effect of using invokedynamic is that the JVM can identify the dispatch logic and know that's what it is

This means it will be able to make much better cost/benefit calculations with regard to inlining and other optimizations

The method handles are quite versatile, while also being things than can be well optimized

The future: from my side

We've still got loads of optimization work to do from the Perl 6 side - on all backends, not just JVM

Many languages have built compilers specifically for the JVM. The Rakudo Perl 6 compiler, by contrast, is multi-backend.

Get to re-use lots, but needs very careful design of abstractions in order to push enough knowledge down to the level where we can use it to exploit things like invokedynamic

The future: invokedynamic

The invokedynamic and the MethodHandles implementations are improving

I hit many bugs with these in JDK7. In fact, early versions segfault on stuff that we're using.

JDK8 should be much more robust

Re-worked invokedynamic implementation can learn from JDK7 issues, but also it's going to be used for Java 8 lambdas, and has Nashorn as a customer

Closing thoughts

Most code, whether written in a static or dynamic language, has fairly static and monomorphic semantics at runtime

The JVM had already got good at optimizing away the cost of certain kinds of dynamic behavior

With invokedynamic, we have an extensible mechanism for helping it optimize away dynamism specific to other languages or problem domains

Thank you!

Hunt me down...

Email: jonathan@edument.se
Twitter: @jnthnwrthngtn

Questions?

P.S. Think I'd be fun to work with? Edument is hiring. Not for invokedynamic work...but if you like teaching/mentoring and building quality stuff, come and say hi. kthx. ☺