Runtime metaprogramming via `java.lang.invoke.MethodHandle`

© Miguel Garcia, LAMP, EPFL
`http://lamp.epfl.ch/~magarcia`

May 14th, 2012

Abstract

The JVM supports runtime code adaptation (to some degree) with JSR 292, which consists of:

- `invokedynamic`: a facility to determine the target for a callsite by letting runtime code inspect static information about the callsite in question; and
- `MethodHandle`s [5], type-safe function pointers that can be combined at runtime into larger units of functionality.

Method handles can be used as a more lightweight replacement for closures, avoiding reification yet capturing lexical context (via `MethodHandle.bindTo()`), not unlike the planned proceduralitzation[1] of lambdas in Java 8.

However these notes explore the pros and cons of method handles for another purpose: runtime specialization of generic code to operate on primitive types, thus avoiding repeated boxing/unboxing. Currently `scalac` relies on static techniques to solve this problem [2].

Source code of the prototype described in these notes can be found at `https://github.com/magarciaEPFL/MethodHandleUtils`.

1 Background

Method handles provide better performance than `j.l.reflect.Method` because (a) security checks are carried out once on creation and not per access; and also because (b) autoboxing can be side-stepped if so desired. For example, the `iadd` instruction to add two `int`s can be represented as a `MethodHandle`:

```
public static int addII(int  a, int  b) { return a + b; }

int result = (int) mh.invokeExact(1, 2);
```

where `mh` above targets `addII` and is invoked without autoboxing (any other method with descriptor `(II)I` bound to `mh` would equally do). Type checking takes place at runtime, and method handles can be bound to targets either using immediate constants (`ldc` instruction) or by means of `MethodHandles.Lookup`.

Method handles are amenable to composition at runtime [5] in a manner that hides concrete method signatures. For example, assignment between compatible

[1] `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html`

types can be expressed as another `MethodHandle` (details in Sec. 2) provided that the setter and getter used for that purpose "match up" at runtime:

```
public static MethodHandle assignment(MethodHandle lhs, MethodHandle rhs) {
    assert isSetter(lhs);
    assert isGetter(rhs);
    return application(lhs, new MethodHandle[] { rhs });
}
```

In terms of runtime specialization (Sec. 3) combiners such as `assignment` above will be used in *specialization factories*, which take as input *tags* denoting the *JVM sort* of type parameters. In the case of the ASM bytecode manipulation library[2] a sort is denoted by an `int` for easy switching:

```
public static final int VOID = 0;
public static final int BOOLEAN = 1;
. . .
public static final int ARRAY = 9;
public static final int OBJECT = 10;
```

In the example of adding two numeric values (both `int`s or both `float`s) the invokers would be:

```
// returns an int
(int)(addOp_SpzFactory(intTag).invokeExact(1, 2))

// returns a float
(float)(addOp_SpzFactory(floatTag).invokeExact(1.0f, 2.0f))
```

2 Building blocks

The utility functions in this section are molded after AST node types in `scalac`. For example:

```
public static MethodHandle ifExpr(MethodHandle cond, MethodHandle thenPart, MethodHandle elsePart) {
    assert isArgless(cond) && isBooleanValued(cond);
    assert isArgless(thenPart);
    assert isArgless(elsePart);
    assert isVoidValued(thenPart) == isVoidValued(elsePart); // TODO test whether both branches have common lub.
    return guardWithTest(cond, thenPart, elsePart);
}
```

That was easy given there's a close enough counterpart in JSR 292 (`guardWithTest()`). But it already showcases the technique at play: method handles given as input become leaves of the tree node (another method handle) the utility function returns. This can also be seen at play when composing a *block expression*:

```
public static MethodHandle blockExpr(MethodHandle[] statements, MethodHandle expr) {
    for(int idx = 0; idx < statements.length; idx++) {
        assert isArglessVoid(statements[idx]);
    }
    assert(isArgless(expr));

    MethodHandle result = null;
```

[2] http://asm.ow2.org/

Listing 1: Sec. 3

```
public static MethodHandle arraySetterUnbound(Class componentClazz) {
    Class arrayClazz = java.lang.reflect.Array.newInstance(componentClazz, 0).getClass();
    return arrayElementSetter(arrayClazz);
}

public static MethodHandle arrayElemSetter(MethodHandle arrRef) {
    Class componentClazz = evalType(arrRef).getComponentType();
    MethodHandle rcvless = arraySetterUnbound(componentClazz);
    MethodHandle result = application(rcvless, new MethodHandle[] { arrRef });
    assert result.type().parameterCount() == 2;
    assert result.type().parameterType(0) == int.class;
    assert result.type().parameterType(1) == evalType(arrRef);
    return result;
}
```

```
    switch(statements.length) {
        case 0 : result = expr; break;
        case 1 : result = foldArguments(expr, statements[0]); break;
        default: result = foldArguments(expr, blockStmt(statements)); break;
    }
    assert isArgless(result);
    assert evalType(result) == evalType(expr);
    return result;
}
```

3 Datatype-agnostic bytecode

Once put together, code that operates on primitive datatypes should run faster than its non-specialized counterpart. For the time being, JIT compilers do not inline method handles as aggressively as required to perform competitively against manually-specialized code. Additionally, combining method handles at runtime also incurs overhead. Even with dedicated inlining by the VM, an approach based on method handles backfires when the running time of the resulting code is short (say, straight line code).

Therefore, the running example below serves as proof-of-concept, and as testbed for JIT compilers supporting JSR 292.

The code in specialization factories cannot itself depend on those datatypes. We saw this in the case of `assignment`, that can be used to compose assignments where LHS and RHS both operate on `int`s, or both operate on `float`s, etc. In general, accessors to datatype-dependent values and locations are reified as `MethodHandle` instances.

Before JIT-ing, data accesses are thus mediated by method handles. That can be relatively fast for field accesses: (there's dedicated API in the form of `Lookup.getGetter()` and `Lookup.getSetter()`) or array accesses, but accesses to method-local variables has to be simulated as shown below (there's no address-of-location as on the CLR Sec. 4.1). These mechanisms are depicted by the snippets in Listing 2 (local vars, simulated using field accesses) and Listing 1 (array accesses).

Listing 2: Sec. 3

```
public static abstract class LocalVar {
    MethodHandle getter = null;
    MethodHandle setter = null;
}

public static class FloatLocalVar extends LocalVar {
    float v = 0;

    public FloatLocalVar() {
        try {
            getter = lookup().findGetter(FloatLocalVar.class, "v", float.class).bindTo(this);
            setter = lookup().findSetter(FloatLocalVar.class, "v", float.class).bindTo(this);
        } catch (Exception e) {
            throw new Error(e);
        }
    }
}
```

Those instuctions that do not depend on runtime-dependent method signatures need not be reified into trees of method handles. However in our running example we reify everything (on the grounds that, if a VM shows good performance under these circumstances, it will work even better when only datatype-dependent instructions are reified.)

The running example adds up the numbers in an input array (say, all `int`s, or all `float`s). The AST-building code for it (covering `int`s only, minor modifications to cover `float`s too) looks like:

```
public static MethodHandle summationMaker(MethodHandle aref) {

    MethodHandle arr_g = arrayElemGetter(aref);
    MethodHandle arr_s = arrayElemSetter(aref);

    IntLocalVar idx = new IntLocalVar();
    IntLocalVar acc = new IntLocalVar();

    MethodHandle line0A = assignment(idx.setter, intConstant(0));
    MethodHandle line0B = assignment(acc.setter, intConstant(0));

    MethodHandle astLength = application(mh_array_length, aref.asType(methodType(java.lang.Object.class)));
    MethodHandle line1_cond = application(mh_intLessThan, idx.getter, astLength);

    MethodHandle body_A1 = application(mh_addII, acc.getter, application(arr_g, idx.getter) );
    MethodHandle body_A = assignment(acc.setter, body_A1);
    MethodHandle body_B = assignment(idx.setter, application(mh_addII, idx.getter, intConstant(1)));

    MethodHandle body = blockStmt( new MethodHandle[] { body_A, body_B });

    MethodHandle line1 = whileLoop(line1_cond, body);

    MethodHandle result = blockExpr(new MethodHandle[] { line0A, line0B, line1 }, acc.getter );

    return result;
}
```

4 Further information

There's not much in the way of tutorials for the JSR-292 API but the following can help to get started:

- `http://medianetwork.oracle.com/video/player/1041168645001`
- `http://code.google.com/p/jsr292-cookbook/`

Tracing VMs:

- Trace-based just-in-time type specialization for dynamic languages [3]
- SPUR: a trace-based JIT compiler for CIL [1]
- Trace-based compilation for the Java HotSpot virtual machine [4]

4.1 Comparison with the CLR

There's a plethora of mechanisms on the CLR for runtime code generation:

- `DynamicMethod`, `http://www.wintellect.com/CS/blogs/krome/archive/2011/03/07/getting-to-know-dynamicmethod.aspx`
- Using `MethodRental.SwapMethodBody` to do Method Level JIT Compilation, `http://weblog.ikvm.net/PermaLink.aspx?guid=977499e8-0a70-4744-9482-5b6132504055`

The CLR term *delegate* may refer to (1) a delegate type (a subclass of `System.Delegate`), or (2) an instance of a delegate type. There's a public constructor,

```
// delegate-type-specific constructor (for a delegate called 'Function' in the example)
public Function(object @object, IntPtr method); /*- native int pointer, a function pointer! */
```

but oftentimes the individual delegate instances thus created are aggregated for multicast purposes using runtime-managed factory methods that return new delegate instances. For example, adding a method reference to a multicast delegate instance.

In the surface syntax of C# a location having a delegate type can be assigned:

- a static method with compatible formal parameters and return type,
- an *`object.instanceMethod`* selector where the method has formals and return type compatible with the LHS delegate,
- an *anonymous function* (a.k.a. closure, e.g. `(double x) => x * 2.0`)

In summary, a delegate instance is to be used like a `scala.FunctionX` instance, yet its actual type is not generic, does not extend any .NET type with a strongly typed `Invoke()` method, and in fact due to nominal subtyping is not the same type as some other delegate taking the same formals to the same return type.

References

[1] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. Spur: a trace-based jit compiler for cil. *SIGPLAN Not.*, 45(10):708–725, October 2010.

[2] Iulian Dragos and Martin Odersky. Compiling generics through user-directed type specialization. In *ICOOOLPS '09: Proc. 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 42–47, New York, NY, USA, 2009. ACM. `http://infoscience.epfl.ch/record/150134/files/p42-dragos.pdf`.

[3] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009.

[4] Christian Häubl and Hanspeter Mössenböck. Trace-based compilation for the java hotspot virtual machine. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 129–138, New York, NY, USA, 2011. ACM.

[5] John R. Rose. Bytecodes meet combinators: invokedynamic on the jvm. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 2:1–2:11, New York, NY, USA, 2009. ACM. `http://cr.openjdk.java.net/~jrose/pres/200910-VMIL.pdf`.