ORACLE®

Lambda Forms: IR for Method Handles

John R. Rose
Da Vinci Machine Project, JSR 292 Lead

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Method Handles in JDK 7: The Good

- Flexible and powerful.
  - Competent to alias any “invoke” instruction.
  - Able to express all functional argument transformations.
- MH graphs are aggressively inlined and optimized.
  - When rooted at invokedynamic.
  - When a constant in a final field.
- Successfully used in multiple projects.

ORACLE

Method Handles in JDK 7: Not-so-good

- “Performance cliff” when inlining does not occur
  - When method handle graph too big (application scale)
  - Or, on invocation of non-constant method handle (!)
- On-the-fly conversion path (generic invoke) is slow
  - Implementation is awkward and complex
- `NoClassDefFoundError` in large applications
  - Due to ad hoc translation of MH graph to bytecodes in JVM
  - Bytecodes are the wrong IR! (Nominal method references)

ORACLE

Method Handles in JDK 7: The Ugly

- MH graph semantics defined as mini-interpreter
  - Hand-written in assembly code (difficult to port, 100s of lines)
  - Argument transforms are defined in terms of interpreter stack
- Therefore, no general fast path for compiled code (!)
- JVM is entangled in MH operations
  - MH has assembly-code pointer installed by JVM
  - JNI native function required when creating every MH node

ORACLE

Rendering MHs to bytecodes (JDK 7)

```
mh = MH.filterReturnValue(A::f, B::g)
    ⇒
return B.g(A.f(param))
    ⇒
0 iload_0
1 invokestatic A.f(int)long
4 invokestatic B.g(long)float
7 freturn
```

- Must use a class loader that can see both A and B.
- What if there is no such class loader?
- What if A and B have the same name spelling??
- How do signature constraints interact???

ORACLE

Bytecode rendering *only for constants*

- Non-constant invocation goes “off the cliff”
  - Into assembly code
- Assembly code is inscrutable to compiler
- Includes special “ricochet frames” (mini-interpreter)
- Compiled-to-compiled calls copy arguments
  - Several times: C2I, MH transform, I2C

ORACLE

JVM Entanglement

- Every method handle node is created with a JNI call.
  - One node for each individual argument transformation.
    (ex: swap, dup, drop, insert, box, unbox, cast)
  - JVM is responsible for mapping transform to assembly stub
  - JDK is responsible for knowing the repertoire of transforms
  - JDK composes low-level transforms (AdapterMethodHandle)
  - JVM decorates them with assembly code handlers
- ⇒ too many cooks in the kitchen

ORACLE

JVM Dis-entanglement

- Root problem: MH chains are too low-level
- The MH chain is the de facto IR in JDK 7.
  - Nodes are low-level argument transformations.
  - At the level of single interpreter instructions.
- Solution: Better IR.
  - At the level of JVM methods.
  - Meshes better with the JVM execution engine.
  - Interprets and/or compiles.
- More representation decisions decoupled from JVM.
- Impact on source base:
  - JDK LOC: 7.0k added, 3.4k deleted, net +3.6k
  - JVM LOC: 5.0k added, 12.6k deleted, net -7.6k

Key IR requirements

- Easy to create “units of behavior”
  - Assemble in pure Java code; simple pointer pasting
- Able to compose reusable building blocks
  - Structure should be inherently reusable & cacheable
- Can be reused frequently (reduced type system)
- Competent to represent method handles
  - Represent all adapters and argument transforms
  - Represent uses of all methods, fields, and constructors
- Similar to JVM methods
  - Missing features ok (minimal control flow)
  - Locally verifiable when rendered to bytecodes
  - Trusted (potentially unsafe) at the edges between blocks

Inspiration

```
λ f g x . f(g(x))

(lambda (f g x)
  (define a (g x))
  (define b (f a))
  b)
```

ORACLE

Example 1: swap arguments

```
mh1 = MHs.permuteArguments(mh0,
      (Object,Object)->Object, {1,0})

mh1 = new SimpleMethodHandle(
λ (mh1:L; x:L, y:L) {
  z:L = MH::invokeBasic(#(mh0); y, x);
  return z
})
```

Lambda Form IR (in one page)

- A LambdaForm is a linear array of Names.
  - First formals, then expressions.
- An expression is a NamedFunction with arguments.
  - Named function is a symbolic reference *on Boot Class Path*.
  - Argument array contains (previous) Names and/or Objects.
  - Calls can be either strongly or weakly typed.
- arity, return value represented as small ints
- no symbolic names (just local Name pointers)
- no control flow (except early exit), so trivially SSA

Lambda Form AST interpreter

```
@Hidden
/** Interpretively invoke this form on the given arguments. */
Object interpretWithArguments(Object... argumentValues) throws Throwable {
    if (TRACE_INTERPRETER)
        return interpretWithArgumentsTracing(argumentValues);
    checkInvocationCounter();
    assert(arityCheck(argumentValues));
    Object[] values = Arrays.copyOf(argumentValues, names.length);
    for (int i = argumentValues.length; i < values.length; i++) {
        values[i] = interpretName(names[i], values);
    }
    return (result < 0) ? null : values[result];
}

@Hidden
/** Evaluate a single Name within this form, applying its function to its arguments. */
Object interpretName(Name name, Object[] values) throws Throwable {
    if (TRACE_INTERPRETER)
        traceInterpreter("| interpretName", name.debugString(), (Object[]) null);
    Object[] arguments = Arrays.copyOf(name.arguments, name.arguments.length, Object[].class);
    for (int i = 0; i < arguments.length; i++) {
        Object a = arguments[i];
        if (a instanceof Name) {
            int i2 = ((Name)a).index();
            assert(names[i2] == a);
            a = values[i2];
            arguments[i] = a;
        }
    }
    return name.function.invokeWithArguments(arguments);
}
```

LF type system

```
λ (a0:L, ..., a9:J) {
  t10:I = nf10(...); ... t19:D = nf19(...);
  return t19 }
```

- basic value type is one of { ref, int, long, float, double }
- (represented as signature letters "LIJFD")
- method type composed of the above, plus void ("V")
- GC-safe, weakly typed
  - trusted, private to java.lang.invoke
- allows rendering to verifiable bytecodes
  - (if conversions are added)

ORACLE

What's in a Name?

- No symbols, just compact small indexes
  - no lexical contours, no non-local references.
- NamedFunction plus a sequence of arguments
  - Object[] arguments
- NF is a symbolic reference to a BCP method
  - can be static, virtual, etc.
  - realized by an arbitrary Method Handle
  - (this part of the design is meta-circular)
- each argument is a previous Name (in same LF)
  - or else an arbitrary constant, boxed as an Object

Lambda Form Execution

- Given a set of arguments and a LambdaForm
  - Allocate an associated value array, one for each Name.
  - Associate incoming arguments with formal Names.
- For each expression, execute the expression.
  - That is, apply the named function to its argument array.
  - The argument array can contain embedded Names.
  - Those names are replaced by their associated values.
- Record each expression value in the value array.
- Return the value associated with the last Name.
  - (It could be another of the associated values, actually.)

Lambda Form AST interpreter

```
@Hidden
/** Interpretively invoke this form on the given arguments. */
Object interpretWithArguments(Object... argumentValues) throws Throwable {
    if (TRACE_INTERPRETER)
        return interpretWithArgumentsTracing(argumentValues);
    checkInvocationCounter();
    assert(arityCheck(argumentValues));
    Object[] values = Arrays.copyOf(argumentValues, names.length);
    for (int i = argumentValues.length; i < values.length; i++) {
        values[i] = interpretName(names[i], values);
    }
    return (result < 0) ? null : values[result];
}

@Hidden
/** Evaluate a single Name within this form, applying its function to its arguments. */
Object interpretName(Name name, Object[] values) throws Throwable {
    if (TRACE_INTERPRETER)
        traceInterpreter("| interpretName", name.debugString(), (Object[]) null);
    Object[] arguments = Arrays.copyOf(name.arguments, name.arguments.length, Object[].class);
    for (int i = 0; i < arguments.length; i++) {
        Object a = arguments[i];
        if (a instanceof Name) {
            int i2 = ((Name)a).index();
            assert(names[i2] == a);
            a = values[i2];
            arguments[i] = a;
        }
    }
    return name.function.invokeWithArguments(arguments);
}
```

Lambda Form interpreter, in one page

```
@Hidden
/** Interpretively invoke this form on the given arguments. */
Object interpretWithArguments(Object... argumentValues) throws Throwable {
    if (TRACE_INTERPRETER)
        return interpretWithArgumentsTracing(argumentValues);
    checkInvocationCounter();
    assert(arityCheck(argumentValues));
    Object[] values = Arrays.copyOf(argumentValues, names.length);
    for (int i = argumentValues.length; i < values.length; i++) {
        values[i] = interpretName(names[i], values);
    }
    return (result < 0) ? null : values[result];
}

@Hidden
/** Evaluate a single Name within this form, applying its function to its arguments. */
Object interpretName(Name name, Object[] values) throws Throwable {
    if (TRACE_INTERPRETER)
        traceInterpreter("| interpretName", name.debugString(), (Object[]) null);
    Object[] arguments = Arrays.copyOf(name.arguments, name.arguments.length, Object[].class);
    for (int i = 0; i < arguments.length; i++) {
        Object a = arguments[i];
        if (a instanceof Name) {
            int i2 = ((Name)a).index();
            assert(names[i2] == a);
            a = values[i2];
            arguments[i] = a;
        }
    }
    return name.function.invokeWithArguments(arguments);
}
```

Lambda Form interpreter, in one page

```
@Hidden
/** Interpretively invoke this form on the given arguments. */
Object interpretWithArguments(Object... argumentValues) throws Throwable {
    if (TRACE_INTERPRETER)
        return interpretWithArgumentsTracing(argumentValues);
    checkInvocationCounter();
    assert(arityCheck(argumentValues));
    Object[] values = Arrays.copyOf(argumentValues, names.length);
    for (int i = argumentValues.length; i < values.length; i++) {
        values[i] = interpretName(names[i], values);
    }
    return (result < 0) ? null : values[result];
}

@Hidden
/** Evaluate a single Name within this form, applying its function to its arguments. */
Object interpretName(Name name, Object[] values) throws Throwable {
    if (TRACE_INTERPRETER)
        traceInterpreter("| interpretName", name.debugString(), (Object[]) null);
    Object[] arguments = Arrays.copyOf(name.arguments, name.arguments.length, Object[].class);
    for (int i = 0; i < arguments.length; i++) {
        Object a = arguments[i];
        if (a instanceof Name) {
            int i2 = ((Name)a).index();
            assert(names[i2] == a);
            a = values[i2];
            arguments[i] = a;
        }
    }
    return name.function.invokeWithArguments(arguments);
}
```

ORACLE

Lambda Form interpreter, in one page

```
@Hidden
/** Interpretively invoke this form on the given arguments. */
Object interpretWithArguments(Object... argumentValues) throws Throwable {
    if (TRACE_INTERPRETER)
        return interpretWithArgumentsTracing(argumentValues);
    checkInvocationCounter();
    assert(arityCheck(argumentValues));
    Object[] values = Arrays.copyOf(argumentValues, names.length);
    for (int i = argumentValues.length; i < values.length; i++) {
        values[i] = interpretName(names[i], values);
    }
    return (result < 0) ? null : values[result];
}

@Hidden
/** Evaluate a single Name within this form, applying its function to its arguments. */
Object interpretName(Name name, Object[] values) throws Throwable {
    if (TRACE_INTERPRETER)
        traceInterpreter("| interpretName", name.debugString(), (Object[]) null);
    Object[] arguments = Arrays.copyOf(name.arguments, name.arguments.length, Object[].class);
    for (int i = 0; i < arguments.length; i++) {
        Object a = arguments[i];
        if (a instanceof Name) {
            int i2 = ((Name)a).index();
            assert(names[i2] == a);
            a = values[i2];
            arguments[i] = a;
        }
    }
    return name.function.invokeWithArguments(arguments);
}
```

Lambda Form interpreter, in one page

```
@Hidden
/** Interpretively invoke this form on the given arguments. */
Object interpretWithArguments(Object... argumentValues) throws Throwable {
    if (TRACE_INTERPRETER)
        return interpretWithArgumentsTracing(argumentValues);
    checkInvocationCounter();
    assert(arityCheck(argumentValues));
    Object[] values = Arrays.copyOf(argumentValues, names.length);
    for (int i = argumentValues.length; i < values.length; i++) {
        values[i] = interpretName(names[i], values);
    }
    return (result < 0) ? null : values[result];
}

@Hidden
/** Evaluate a single Name within this form, applying its function to its arguments. */
Object interpretName(Name name, Object[] values) throws Throwable {
    if (TRACE_INTERPRETER)
        traceInterpreter("| interpretName", name.debugString(), (Object[]) null);
    Object[] arguments = Arrays.copyOf(name.arguments, name.arguments.length, Object[].class);
    for (int i = 0; i < arguments.length; i++) {
        Object a = arguments[i];
        if (a instanceof Name) {
            int i2 = ((Name)a).index();
            assert(names[i2] == a);
            a = values[i2];
            arguments[i] = a;
        }
    }
    return name.function.invokeWithArguments(arguments);
}
```

ORACLE

Lambda Form interpreter, in one page

```
@Hidden
/** Interpretively invoke this form on the given arguments. */
Object interpretWithArguments(Object... argumentValues) throws Throwable {
    if (TRACE_INTERPRETER)
        return interpretWithArgumentsTracing(argumentValues);
    checkInvocationCounter();
    assert(arityCheck(argumentValues));
    Object[] values = Arrays.copyOf(argumentValues, names.length);
    for (int i = argumentValues.length; i < values.length; i++) {
        values[i] = interpretName(names[i], values);
    }
    return (result < 0) ? null : values[result];
}

@Hidden
/** Evaluate a single Name within this form, applying its function to its arguments. */
Object interpretName(Name name, Object[] values) throws Throwable {
    if (TRACE_INTERPRETER)
        traceInterpreter("| interpretName", name.debugString(), (Object[]) null);
    Object[] arguments = Arrays.copyOf(name.arguments, name.arguments.length, Object[].class);
    for (int i = 0; i < arguments.length; i++) {
        Object a = arguments[i];
        if (a instanceof Name) {
            int i2 = ((Name)a).index();
            assert(names[i2] == a);
            a = values[i2];
            arguments[i] = a;
        }
    }
    return name.function.invokeWithArguments(arguments);
}
```

ORACLE

Lambda Form interpreter, in one page

```
@Hidden
/** Interpretively invoke this form on the given arguments. */
Object interpretWithArguments(Object... argumentValues) throws Throwable {
    if (TRACE_INTERPRETER)
        return interpretWithArgumentsTracing(argumentValues);
    checkInvocationCounter();
    assert(arityCheck(argumentValues));
    Object[] values = Arrays.copyOf(argumentValues, names.length);
    for (int i = argumentValues.length; i < values.length; i++) {
        values[i] = interpretName(names[i], values);
    }
    return (result < 0) ? null : values[result];
}

@Hidden
/** Evaluate a single Name within this form, applying its function to its arguments. */
Object interpretName(Name name, Object[] values) throws Throwable {
    if (TRACE_INTERPRETER)
        traceInterpreter("| interpretName", name.debugString(), (Object[]) null);
    Object[] arguments = Arrays.copyOf(name.arguments, name.arguments.length, Object[].class);
    for (int i = 0; i < arguments.length; i++) {
        Object a = arguments[i];
        if (a instanceof Name) {
            int i2 = ((Name)a).index();
            assert(names[i2] == a);
            a = values[i2];
            arguments[i] = a;
        }
    }
    return name.function.invokeWithArguments(arguments);
}
```

Lazy Method Handle interpretation

- Initially, direct AST interpretation of MH IR.
- IR can be presented the JVM lazily.
  - Early AST interpretation in Java code.
  - Later insertion into the JVM for direct execution.
- Insertion is (currently) by rendering to bytecodes
  - Uses anonymous class mechanism, as an optimization.
- Mixed mode:
  - AST interpreter (in Java code)
  - ⇒ bytecode interpreter (after lazy
  - ⇒ JIT compilation
  - ⇒ (etc…)

ORACLE

Integrating LF + MH + bytecode

- call from MH to LF = jump to `mh.form`
- call from LF to MH = NamedFunction `invokeBasic`
  - (unchecked version of invokeExact; building block)
- call from LF to arb. Java method: DMH "linkers" (later)
  - DMH = "Direct Method Handle"
- call from arbitrary bytecode to MH: LF adapters
  - Need adapter code to move arguments
  - Also introduces hidden contextual argument (later)

Example 1: swap arguments
(single-use version)

```
mh1 = MHs.permuteArguments(mh0,
      (Object,Object)->Object, {1,0})
```

```
mh1 = new SimpleMethodHandle(
λ (mh1:L; x:L, y:L) {
  z:L = MH::invokeBasic(#(mh0); y, x);
  return z
})
```

Example 2: return a constant value (single-use version)

```
mh1 = MHs.constant(Object, "invariable")

mh1 = new SimpleMethodHandle(
λ (mh1:L) {
  k:L = *ValueConversions::identity(#("invariable"));
  return k
})
```

Bound Method Handles

- Open-ended schema of “struct”-like classes
- Rooted at BoundMethodHandle
  - Each “species” handles one structure layout.
  - Depth = 1: Species inherit from BMH, but are all final.
  - All fields are final (immutable).
- Each species used via a set of method handles
  - Constructor, extender, accessors.
- BMH species are generated as needed.

ORACLE

Example 1B: swap arguments (BMH version)

```
mh1 = new BoundMethodHandle.Species_L(
λ (m; x:L, y:L) {
  mh0:L = BoundMethodHandle::argL0(m);
  z:L = MH::invokeBasic(mh0; y, x);
  return z
}, mh0)
```

Example 2B: return a constant value (BMH version)

```
mh1 = MHs.constant(Object, "invariable")

mh1 = new BoundMethodHandle.Species_L(
λ (m:L) {
  k:L = BoundMethodHandle::argL0(m);
  return k
}, "invariable")
```

ORACLE

Direct Method Handles

- Capability for using one Java method
  - Or field or constructor
  - Implements CONSTANT_MethodHandle constants
- Carries an internal JVM cookie “`MemberName`”
- Performs needed checks or conversions
- Has internal weakly-typed jump to its member-name
  - For methods and constructors, uses a “linker intrinsic”
  - For fields (static & instance), uses sun.misc.Unsafe

Direct Method Handle “Linkers”

- Weakly-typed invocation of arbitrary member-names
- Examples:

```
MH::linkToStatic(#(Thread::current))
MH::linkToStatic(s,j,d,k,l; #(System::arraycopy))
MH::linkToVirtual(obj; #(Object::hashCode))
MH::linkToInterface(cmp,x,y; #(Comparator::compare))
MH::linkToSpecial(str; #(String::length))
MH::linkToSpecial(sb,len; #(StringBuilder::<init>))
```

- Oddity: The member-name is the *trailing* argument
  – Forces caller to perform argument shuffling
  – Trailing argument can be used and transparently dropped
  – Enables compiled fast paths w/o special JVM handling

ORACLE

Example 3: call a regular method (single-use version)

```
mh1 = Object::hashCode
nm1 = (MemberName(Object::hashCode))

mh1 = new SimpleMethodHandle(
λ (m:L, obj:L) {
  v:I = MH::linkToVirtual(obj; #(nm1))
  return v
})
```

Example 3B: call a regular method (real version)

```
mh1 = Object::hashCode
nm1 = (new MemberName(Object::hashCode))

mh1 = new DirectMethodHandle(
λ (m:L, obj:L) {
  n:L = DMH::memberName(m);
  v:I = MH::linkToVirtual(obj; n);
  return v
}, nm1)
```

Metacircularity (bootstrapping)

- LF interpreter can be written down in one page
- LF interpreter uses MHs
  - ... which in turn may use the LFI
- In particular, LFI uses DMHs
  - ...to access things like Class.cast
- But DMHs are defined in terms of LFs!
  - requires DMH methods to work immediately
  - requires BMH accessors *almost* immediately
- Therefore, eager byte-compilation of a few LFs.
- The rest can be managed lazily
  - Amortize costs of sharing (tabulation) and compilation

Bytecode call sites for JSR 292

- Two kinds: `invokedynamic` and "`invokehandle`".
- In both cases, there is a linked contextual argument.
  - For invokedynamic, it is the linked `CallSite` instance.
    - Invocation must insert and invoke the call site target.
  - For method handles, it is the resolved `MethodType` value.
    - Invocation must reify the MT enough to check it.
    - Generic invoke uses the MT to direct arg. conversions.
- We formalize this in the JVM via "appendix" args.
  - Linking indy or MH.invoke makes an up-call to the JDK.
  - The JDK computes a LF and appendix argument.
  - The JVM records *both* and uses them for all calls.

Example 4: Linking `MH.invokeExact`

```
x = mh.invokeExact(12,3.14)

x = A(mh, 12, 3.14; app_MT)
where A = λ (m:L, a1:I, a2:D; mt:L) {
  c:V = *Invokers::checkExactType(m, mt);
  v:L = MH::invokeBasic(m, a1, a2);
  return v
}
where app_MT = #((int,double)->Object))
```

Example 5: Linking `invokedynamic`

```
x = invokedynamic[BSM...](12,3.14)

x = B(12, 3.14; app_CS)
where B = λ (a1:I, a2:D; cs:L) {
  m:L = CallSite::getTarget(cs);
  v:L = MH::invokeBasic(m, a1, a2);
  return v
}
where app_CS = #( result of running BSM )
```

Example 6: function composition

```
mh = MH.filterReturnValue(A::f, B::g)

mh = new BoundMethodHandle.Species_LL(
λ (m; x:I) {
  f:L = BoundMethodHandle::argL0(m);
  y:J = MH::invokeBasic(f; x);
  g:L = BoundMethodHandle::argL1(m);
  z:F = MH::invokeBasic(g; y);
  return z
}, mh0)
```

Status

- Committed to JDK 8 in “hotspot-comp” repository
- Initial feedback is neutral to positive
  - Thanks, MLVM early adopters!
- No sign of `NoClassDefFoundError` anymore.
- Performance cliff has smoothed, apparently.
  - Report: JRuby “tictactoe” test runs 80% faster.

ORACLE

Future work

- Performance tuning
  - Caching (known useful patterns, like DMHs and adapters)
  - Interning (emergent common structures)
  - (Encouraging early results: equal or faster with little tuning)
- Static computation (optional, for some platforms)
- Extended basic block capability (multiple exits)
- Additional type inference, to reduce casting
- Tail-calls (to reduce the “epic” backtraces)
- Use to build Functional Interface Delegate Objects

ORACLE

ORACLE®