Translation of lambda expressions in javac

Brian Goetz, May 2010

1. About this document

This document outlines a strategy for translating portions of the "Lambda Strawman" proposal from Java source code to bytecode. Syntax choices reflect that of the strawman and are for illustrative purposes only. The dark grey boxes reflect the Java code to be translated; the light grey boxes illustrate the decompiled bytecode.

2. JVM dependencies

This document assumes the availability of several features from JSR-292, including method handles and the enhanced "LDC" bytecode that takes a MethodRef constant type and returns a DirectMethodHandle. In the translated examples, a pseudo-syntax for method references will be used which will translate to this LDC operation.

Additionally, we assume that the MethodHandle class will be extended to provide a method asSam(Class sam) which will provide an optimized form of injecting a SAM interface into a method handle.

3. Representation

Function types will be implemented using method handles. Method handles are strongly dynamically typed but are weakly statically typed; in particular they are invariant in that they expect the exact argument types specified in their internal method descriptors.

Function types may or may not be exposed in the source language; in any case migration compatibility requires us to support conversion of lambda expressions to SAM types. Most of the examples in this document will assume SAM-conversion though we will also seek means to reduce the overhead of SAM-conversion where unnecessary.

All lambdas are lowered by the static compiler into static private methods in the class in which they appear.

4. Performance concerns

It is important that the translation of lambda expressions not impose excessive performance costs over the "old way" of expressing the equivalent concept. Performance costs can take multiple forms:

- Additional static classes generated at compile time (as with inner classes)
- Incremental static footprint in enclosing classes
- Additional costs associated with class initialization (static initializers)
- Dynamic memory footprint of lambda capture (method handles, frames)

- Dynamic memory footprint of lambda invocation

5. Type 1 lambdas – “stateless” functions

The simplest form of lambda expression to translate is one that captures no state from its enclosing scope:

```
class A {
    public void foo() {
        List<String> list = ...
        list.forEach( #(String s) { System.out.println(s); } );
    }
}
```

Here, the only state needed to evaluate the lambda is its arguments. (Note that this does not entail that the lambda is side-effect free or idempotent.)

Type 1 lambdas can be translated directly to method handles without any per-capture costs or any additional classes generated. When SAM-converted, they are represented by a statically computed method handle with the appropriate SAM interface injected, which can happen at class initialization time. We’ll assume for illustrative purposes that the forEach() method takes a SAM interface called Block<T>.

```
class A {
    private static Block<T> $sam$1
        = (Block<T>) A#closure$1.asSam(Block.class);

    private static void closure$1(String s) {
        System.out.println(s);
    }

    public void foo() {
        List<String> list = ...
        list.forEach( $sam$1 );
    }
}
```

The source pseudo-syntax “C#functionName” invokes the LDC bytecode that takes a MethodRef and returns a MethodHandle. The not-yet-implemented asSam() method turns that method handle into a method handle that also implements the desired interface.

5.1. Performance analysis

The performance cost of this approach for Type 1 lambdas is quite modest. There is no per-capture cost; the tradeoff is an additional static field in the class, and an additional one or two heap nodes created at class initialization time (whether or not it is ever used). There is no additional footprint or instantiation cost for instances of A.

If the VM can inline forEach() (which can frequently be done if the VM can sufficiently sharpen the type of the incoming list), it can continue to inline through the closure, completely eliminating the runtime cost of both the List and lambda abstractions.

6. Type 2 lambdas – functions capturing final state

The next simplest form of lambda expression involves capture of enclosing final state and/or fields from enclosing instances (we can treat field capture as final capture of one or more enclosing class 'this' references) but no mutable fields.

```
class B {
    public void foo() {
        List<Person> list = ...
        final int bottom = ..., top = ...;
        List inRange = list.filter(
            #(Person p) { p.size >= bottom && p.size <= top } );
    }
}
```

Here, our lambda captures the final local variables bottom and top from the enclosing scope.

6.1. Translating as individual arguments

One approach is to construct the signature of the lowered method by prepending all the captured finals to the free variables, as shown in translation B1. At lambda capture time, one of the method handle combinators (such as insertArgs or bindTo) can be used to package these from the enclosing scope, yielding a method handle whose descriptor matches that of the lambda it corresponds to.

```
class B1 {
    private static boolean closure$1(int bottom, int top,
                                     Person p) {
        return p.size >= bottom && p.size <= top;
    }

    public void foo() {
        List<Person> list = ...
        final int bottom = ..., top = ...;
        Filter<Person> filter = (Filter<Person>)
            MethodHandles.insertArgs(B1#closure$1, bottom, top)
            .asSam(Filter.class);
        List inRange = list.filter(filter);
    }
}
```

Alternately, the construction of the method handle could use a repeated chain of MethodHandle.bindTo() calls, one for each captured final, which would have the advantage that the JVM would do partial application (and therefore enable optimizations such as dead code elimination, null check elimination, range check elimination, etc) specialized to the specific captured finals. Currently both of these translations (insertArgs vs repeated bindTo) have their share of per-capture heap overhead, including varargs handling and boxing. Close work with the JSR-292 expert group will be needed needed to address this.

6.2. Translation as inner class

A second approach is to generate an inner class to handle both captured finals and SAM wrapping, as illustrated by translation B2. Here, method handles are not used at all; instead, a class is generated for the closure, which provides final fields for any captured finals, and also implements the appropriate SAM interface. An instance of this class is created at the point of capture.

```
class B2 {
    private static class Closure$1 implements Filter<Person> {
        private final int bottom, top;

        Closure$1(int bottom, int top) {
            this.bottom = bottom;
            this.top = top;
        }

        private static boolean closure$1(Person p) {
            return p.size >= bottom && p.size <= top;
        }
    }

    public void foo() {
        List<Person> list = ...
        final int bottom = ..., top = ...;
        List inRange = list.filter(new Closure$1(bottom, top));
    }
}
```

6.3. Translation using a frame class

A third approach for Type 2 lambdas includes using a Frame class to store the captured final locals. (See section Frame Optimizations below regarding techniques for reducing the static footprint overhead of the generated Frame class.)

```
class B3 {
    private static class Frame$1 {
        private final int bottom, top;

        Frame$1(int bottom, int top) {
            this.bottom = bottom;
            this.top = top;
        }
    }

    private static boolean closure$1(Frame$1 frame,
                                     Person p) {
        return p.size >= frame.bottom && p.size <= frame.top;
    }

    public void foo() {
        List<Person> list = ...
        final int bottom = ..., top = ...;
        Filter<Person> filter = (Filter<Person>)
            B3#closure$1.bindTo(new Frame(bottom, top))
            .asSam(Filter.class);
        List inRange = list.filter(filter);
    }
}
```

6.4. Performance analysis

The three approaches have different static and dynamic costs. Estimating and comparing the costs very quickly gets down into the performance characteristics of the MethodHandle implementation.

A naïve analysis of option B1 (with insertArgs) yields boxing the top and bottom values into Integers, boxing them into a varargs array, and creating a heap node to wrap the underlying method handle and hold a reference to that array – which could be as many as five heap objects per lambda capture (assuming the asSam operation is done with interface injection). Generalizing in the number of captured arguments, this comes down to $3 + n_b$, where n_b is the number of captured primitives requiring boxing.

A naïve analysis of option B1 with repeated .bindTo calls requires boxing each of the primitives (as in the previous example) and potentially an additional node per injected argument, but does not require the varargs array, for a total of $2 + n + n_b$ heap nodes.

Option B2 is much simpler to analyze; one additional (nominal) class generated per lambda, plus one additional object per lambda capture.

A naïve analysis of Option B3 requires no boxing of primitives or varargs creation but does require instantiation of a Frame object to hold the primitives, for a likely total of 2-3 heap objects per lambda capture, plus potentially the additional Frame class generated (but see section Frame Optimizations below.)

If the VM can inline through the call taking the closure (and likely into the closure), things get quite a bit better – the primitive boxing goes away, the varargs boxing / frame goes away, and the method handles go away, driving the dynamic instantiation and footprint cost to zero.

6.5. Possible method handle optimizations

The current API for method handles seems to be tilted towards the case of only binding to (partially applying) a single leading argument. If this restriction were relaxed, and the bindTo() call were a signature-polymorphic methods that could take primitives without boxing, many of the intermediate heap nodes and boxings from case B2 could be elided. Similarly, since bindTo() and asSam() are always called together, combining these into a single call is also likely to be a win.

7. Type 3 lambdas – functions capturing mutable state

The most complicated translation is required for lambda expressions that capture mutable local variables from the enclosing context. This entails not only making access to that state available to the lambda, but rewriting the code in the enclosing context.

```
class C {
    public void foo() {
        List<String> list = ...
        shared int totalSize = 0;
        list.forEach( #(String s) { totalSize += s.length() } );
        System.out.println(totalSize);
    }
}
```

In such a situation, the “local” variable totalSize is shared between the body of the method foo() and the lambda expression. The lifetime of the totalSize variable is the longer of the lifetime of the foo() invocation and the lifetime of the lambda expression created in foo().

We implement this by hoisting the totalSize variable into a frame object, and rewrite both the lambda and the body of foo() to reference totalSize as a field of the frame.

```
class C {
    private static class Frame$1 {
        private int totalSize;
    }

    private static boolean closure$1(Frame$1 frame, String s) {
        frame.totalSize += s.length();
    }

    public void foo() {
        List<String> list = ...
        Frame$1 frame = new Frame$1();
        frame.totalSize = 0;
        Block<String> block = (Block<String>)
            C#closure$1.bindTo(frame)
            .asSam(Block.class);
        list.forEach(block);
        System.out.println(totalSize);
    }
}
```

In Type 3, we may also have to deal with captured finals, just as in Type 2. We would have to pick one of the mechanisms outlined in the previous section for handling the captured finals, any of which is compatible with the frame approach outlined here.

7.1. Performance analysis

The costs associated with type 3 lambdas are fairly similar to strategy "B3" for Type 2. Each lambda capture allocates a frame and a SAM injection wrapper around a likely-shared method handle. Just as with Type 2, these costs disappear under full inlining.

8. Frame Optimizations

One of the apparent costs of the Type 3 case (and option B3 of the Type 2 case) is that we generate an additional nominal class that is lambda-specific. However, in practice, most of these classes can be eliminated.

Most lambdas capture only a few items from their enclosing scope. We can therefore predefine a number of utility Frame classes, such as those holding a single integer, two integers, one reference, two references, an integer and a reference, etc. (We can further reduce the number of required cases by "erasing" primitives such as boolean, byte, char, and short into an int.) A dozen or so classes would probably cover most of the common cases, especially as it is possible to use a bigger frame than necessary (for example, one might predefine frame classes with 1, 2, 3, and 5 slots, and use the 5-slot frame for 4-slot closures.)

Similarly, arrays can also be used as frames if the types of the captured variables are compatible, so if a lambda were to capture four ints only, an array of four ints could be used as a frame.

The choice of frame representation is completely local and can be changed from one compiler version to another without violating compatibility, since only method handles and SAM classes ever cross class boundaries.

9. Capturing from multiple scopes

Hoisting captured locals into frames is made more complicated when the locals are defined in different scopes.

```
class D {
    public void foo() {
        shared int grandTotal = 0;
        List<Department> departments = ...
        List<Employee> employees = ...
        departments.forEach{ #(Department d) {
            shared int deptTotal = 0;
            employees
                .filter(#(Employee e) { e.dept == d })
                .forEach(#(Employee e) {
                    deptTotal += e.salary;
                    grandTotal += e.salary;
            });
            System.out.printf("Total for dept %s = %d",
                              d, deptTotal);
        };
        System.out.printf("Grant total = %d", grandTotal);}
    }
}
```

In the inner forEach lambda, both the deptTotal and grandTotal variables are captured, from different scopes. We cannot simply hoist the deptTotal variable into the scope outside the outer forEach(), since for each iteration of the outer forEach(), deptTotal is a *different variable.* (When capturing finals there is no problem here if they are from different scopes as the values can be copied at the point of capture.)

Here, we create a separate frame for each scope, and link inner frames back to outer frames (as if the inner frames were actually inner classes.)

```
    private static class Frame$1 {
        int grandTotal;

        private Frame$1(int grandTotal) {
            this.grandTotal = grandTotal;
        }
    }

    private static class Frame$2 {
        Frame$1 outer;
        int deptTotal;

        private Frame$2(Frame$1 outer, int deptTotal) {
            this.outer = outer;
            this.deptTotal = deptTotal;
        }
    }
```

As we enter a scope with captured mutable variables, we instantiate a frame for that scope (even if we are not sure that the lambda will be captured.) Any final variables captured by a lambda within a lambda must be treated as being captured by the outer

lambda, so they can be provided to the inner lambda (since the outer lambda is responsible for creating and binding the inner lambda.)

```
public class D {

    private static void closure$1(Frame$1 frame$1,
                                  List<Employee> employees,
                                  Department d) {
        Frame$2 frame$2 = new Frame$2(frame$1, 0);
        employees
            .filter(MethodHandles
                    .insertArguments(D#closure$1, 0, d)
                    .asSam(Filter.class))
            .forEach(MethodHandles
                    .insertArguments(D#closure$3, 0, frame$2)
                    .asSam(Block.class));
        System.out.printf("Total for dept %s = %d",
                          d, frame$2.deptTotal);
    }

    private static boolean closure$2(Department d, Employee e) {
        return e.dept == d;
    }

    private static boolean closure$3(Frame$2 frame, Employee e) {
        frame.deptTotal += e.salary;
        frame.outer.grandTotal += e.salary;
    }

    public void foo() {
        Frame$1 frame$1 = new Frame$1(0);
        List<Department> departments = ...
        List<Employee> employees = ...

        departments.forEach(MethodHandles
            .insertArguments(D#closure$2, 0, frame$1)
            .asSam(Block.class));
        System.out.printf("Grand total = %d", frame$1.grandTotal);
    }
}
```

10. Multiple lambdas per scope

It is quite possible that a single scope may have multiple lambdas, each of which captures a different subset of the mutable variables in that scope. In that case there would be a single frame class generated for the *scope*, which would be shared by all the lambdas in that scope.