From Lambdas to Bytecode

Brian Goetz
Java Language Architect

SAM conversion

- Lambda expressions are anonymous methods
  - Always converted to "SAM" (single abstract method) types

```
interface Predicate<T> { boolean apply(T t); }

Collection<T> filter(Predicate<T> p) { ... }

kids = people.filter(#{ p -> p.age < 18 });
```

- Compiler takes care of type inference and SAM target selection
  - Figures out that the lambda can be converted to Predicate<Person>
- But then, what bytecode should the compiler emit?

Translation options

- Could just translate to inner classes
  - `#{ p -> p.age < TARGET }` translates to
    ```
    class Foo$1 implements Predicate<Person> {
        private final int v0;
        Foo$1(int $v0) { this.$v0 = v0 }
        public boolean apply(Person p) {
            return (p.age < $v0);
        }
    }
    ```
  - Capture == invoke constructor (`new Foo$1(TARGET)`)
  - One class per lambda expression – yuck
  - Would burden lambdas with identity
    - Would like to improve performance over inner classes
  - Why copy yesterday's mistakes?

Translation options

- Could translate directly to method handles
  - Desugar lambda body to a static method
  - Capture == take method reference + curry captured args
  - Invocation == MethodHandle.invoke
- Whatever translation we choose becomes not only implementation, but a binary specification
  - Want to choose something that will be good forever
  - Is the MH API ready to be a permanent binary specification?
  - Are raw MHs yet performance-competitive with inner classes?

Translation options

- What about “inner classes now and method handles later”?
  - But old class files would still have the inner class translation
  - Java has never had “recompile to get better performance” before
- Whatever we do now should be where we want to stay
  - But the “old” technology is bad
  - And the “new” technology isn’t proven yet
  - What to do?

Invokedynamic to the rescue!

- We can use invokedynamic to delay the translation strategy until runtime
  - Invokedynamic was originally intended for dynamic languages, not statically typed languages like Java
  - But why should the dynamic languages keep all the dynamic fun for themselves?
- We can use invokedynamic to embed a *recipe* for constructing a lambda at the capture site
  - At first capture, a translation strategy is chosen and the call site linked
  - Subsequent captures bypass the slow path
  - As a bonus, stateless lambdas translated to constant loads

Layers of cost for lambdas

- Any translation scheme imposes costs at several levels:
  - Linkage cost – one-time cost of setting up capture
  - Capture cost – cost of creating a lambda
  - Invocation cost – cost of invoking the lambda method
- For inner class instances, these correspond to:
  - Linkage: loading the class
  - Capture: invoking the constructor
  - Invocation: invokeinterface
- The key cost to optimize is *invocation* cost

Code generation strategy

- All lambda bodies are desugared to static methods
  - For “stateless” (non-capturing) lambdas, lambda signature matches SAM signature exactly

```
#{ String s -> s.length() == 10 }
```

  - Becomes (when translated to Predicate<String>)

```
static boolean lambda$1(String s) {
    return s.length() == 10;
}
```

Code generation strategy

- For lambdas that capture variables from the enclosing context, these are prepended to the argument list
  - We only allow capture of (effectively) final variables
  - So we can freely copy variables at point of capture

```
#{ String s -> s.length() == target }
```

  - Becomes (when translated to Predicate<String>)

```
static void lambda$1(int target, String s) {
    return s.length() == target;
}
```

Code generation strategy

- At point of lambda capture, compiler emits invokedynamic call to create SAM ("lambda factory")
  - Call arguments are the captured values (if any)
  - Bootstrap is method in language runtime ("metafactory")
  - Static arguments identify properties of the lambda and SAM

  ```
  list.filter(#{ s -> s.length() == target });
  ```

  Becomes

  ```
  list.filter(indy[bsm=mf, args=...](target));
  ```
- Static args encode properties of lambda and SAM
  - Is lambda cacheable?
  - Is SAM serializable?

Static bootstrap arguments

- Static bootstrap arguments might look like

```
metaFactory(Lookup caller,                 // provided by VM
            String invokedName,            // provided by VM
            MethodType invokedType,        // provided by VM
            Class<?> samClass,             // SAM conversion target
            String samMethodName,          // SAM conversion target
            MethodType samMethodType,      // SAM conversion target
            MethodHandle handle,           // lambda body
            Class<?> implClass,            // lambda body
            String implName,               // lambda body
            MethodType implType,           // lambda body
            String uniqueToken)            // needed for serialization
```

Benefits of invokedynamic

- Invokedynamic is the ultimate lazy evaluation idiom
  - For stateless lambdas that can be cached, they are initialized at first use and cached at the capture site
  - Programmers frequently cache inner class instances (like Comparators) in static fields, but indy does this better
- No overhead if lambda is never used
  - No field, no static initializer
  - Just some extra constant pool entries
- SAM conversion strategy becomes a pure implementation detail
  - Can be changed dynamically by changing metafactory

Possible translation strategies

- Spin inner classes dynamically
  - Generate the same class the compiler would, just at runtime
  - This is likely to be the initial strategy, until we can prove that there's a better one
- Spin per-SAM wrapper classes (one wrapper class per SAM type)
  - Use method handles, for invocation
  - Use ClassValue to cache wrapper for SAM
  - Some annoying interactions with erasure here
- Use dynamic proxies
- Use MethodHandle.asInstance
  - This is basically pushing the problem to the MH runtime
- Use VM-private APIs to build object from scratch

Serialization

- Users will expect this code to work:

```
interface Foo extends Serializable {
    public boolean eval();
}
Foo f = #{ false };
// now serialize f
```

- Since our code generation strategy is dynamic, our serialization strategy must be also
  - Answer: use readResolve / writeReplace
  - Instead of serializing lambda directly, serialize the recipe (say, to some well defined interface SerializedLambda)
  - On deserialization, reconstitute from recipe
    - Using then-current lambda creation strategy, which might be different from the one that originally created the lambda