Java **Intrinsic**s

Rémi Forax – FOSDEM 2014

ARFU.compareAndSet

```
public class Linked {
  private static final AtomicReferenceFieldUpdater<Linked, Node> ARFU =
      AtomicReferenceFieldUpdater.newUpdater(Linked.class, "head", Node.class);
  private volatile Node head;
  private boolean compareAndSet(Node expected, Node newValue) {
    return ARFU.compareAndSet(this, expected, newValue);
  }
  public void addFirst(int element) {
    for(;;) {
      Node head = this.head;
      Node newNode = new Node(element, head);
      if (compareAndSet(head, newNode)) {
        return;
      }
    }
  }
```

sun.misc.Unsafe.compareAndSwap

```
public class Linked {
  private static final Unsafe UNSAFE;
  private static final long OFFSET;

  static { /* initialize UNSAFE and OFFSET */ }

  private volatile Node head;

  private boolean compareAndSet(Node expected, Node newValue) {
    return UNSAFE.compareAndSwapObject(this, OFFSET, expected, newValue);
  }

  public void addFirst(int element) {
    for(;;) {
      Node head = this.head;
      Node newNode = new Node(element, head);
      if (compareAndSet(head, newNode)) {
        return;
      }
    }
  }
```

Concurrency in 9 - JEP

Get ride of sun.misc.Unsafe !

- Unsafe == peek and poke of Java

Problem

- classes of java.util.concurrent.atomic are slow compared to Unsafe.xxx

- Java has no concept of field reference
  - & foo.bar

Inside ARFU.compareAndSet

compareAndSet is slow compared to compareAndSwap due to the erasure of generics

```
public boolean compareAndSet(T obj, V expect, V update) {
   if (obj == null || obj.getClass() != tclass || cclass != null ||
      (update != null && vclass != null &&
       vclass != update.getClass()))
          updateCheck(obj, update);
   return unsafe.compareAndSwapObject(obj, offset, expect, update);

void updateCheck(T obj, V update) {
   if (!tclass.isInstance(obj) ||
       (update != null && vclass != null && !vclass.isInstance(update)))
          throw new ClassCastException();
   if (cclass != null && cclass.isInstance(obj))
        throw new RuntimeException(...)
```

Intrinsic keyword !

```
public class Linked {
  private volatile Node head;
  private boolean compareAndSet(Node expected, Node newValue) {
    return Volatiles.compareAndSet("head", this, expected, newValue);
  }
  ...
}
public class Volatiles {
  public static intrinsic <T> boolean compareAndSet(String fieldName,
                                    Object current, T expected, T newValue) {
    throw new InternalError();
  }
  public static CallSite bootstrap(Lookup lookup, String name,
                                   MethodType type, MethodHandle impl) { ... }
```

In the compiler

```
public class Linked {
  private boolean compareAndSet(Node expected, Node newValue) {
    return Volatiles.compareAndSet("head", this, expected, newValue);
  }
}
```

Should not use declared types as signature !

boolean Volatiles.**compareAndSet**(String, Object, Object, Object)

but propagated types as signature !

boolean Volatiles.**compareAndSet**(String, **Linked**, **Node**, **Node**)

and generate an **invokedynamic**

so the bootstrap method can check the signature at link time !

The bootstrap method

Don't call Volatiles.compareAndSet implementation !

Generate a method handle tree

```
boolean mh(String a0, Linked a1, Node a2, Node a3)
  guardWithTest
    a0 == constant(value of a0 of the first call)
    Unsafe.compareAndSwap(a1, constant(offset(a0)),
                                  a2, a3)
    throw new ISE()
```

Or maybe a slow path ?

At runtime

The JIT will remove the 'if' generated from guardWithTest because the String "head" is constant

Almost **same generated code** that using Unsafe.compareAndSwapObject()

Two supplementary nullchecks that test if expected and newValue are not null

- Glitch in the way LambdaForm impl do downcasting Node -> Object

Intrinsic keyword

Decouple the method call from the callee

Fully typecked and safe

Use invokedynamic and send implementation found by the compiler as method handle

FFI JEP

Bypass JNI, do direct call from Java to C libs

Goal: as fast as a C call ?

- Teach the JIT to generate assembly code (call convention, etc) to do direct call

- Need metadata describing how to do a call to C function

FFI JEP

How to give call Metadata to the JIT ?

- Create a new kind of MethodHandle
  - MethodHandle Lookup.findCFunction(...)

How to transmit this metadata from a declared method to the method handle ?

- Use **intrinsic** !

FFI JEP

Use reflectAs to find the java.lang.reflect.Method of a method handle

```
public class LibC {
  @FFI(...)
  public static intrinsic long getpid() {
     throw new InternalError();
  }

  public static CallSite bootstrap(Lookup lookup, String name,
                                   MethodType type, MethodHandle impl) {
      Method method = MethodHandles.reflectAs(Method.class, impl);
      FFI ffi = method.getAnnotation(FFI.class);
      ...
      return new ConstantCallSite(lookup.findCFunction(...));
    }
}
```

No proxy !!

Conclusion

We need an **intrinsic** syntax in Java

The compiler support is already in 8 !

it will also help

- Dynamic Languages :
  Java --> JRuby bridge, Java --> Nashorn, etc
- TCK :
  can emit some invokedynamic in Java, no ASM !
- CDI : replace proxies by invokedynamic
  prototype on top of Weld by Antoine Sabot-Durand, Red Hat