JSR 292 Cookbook

Rémi Forax

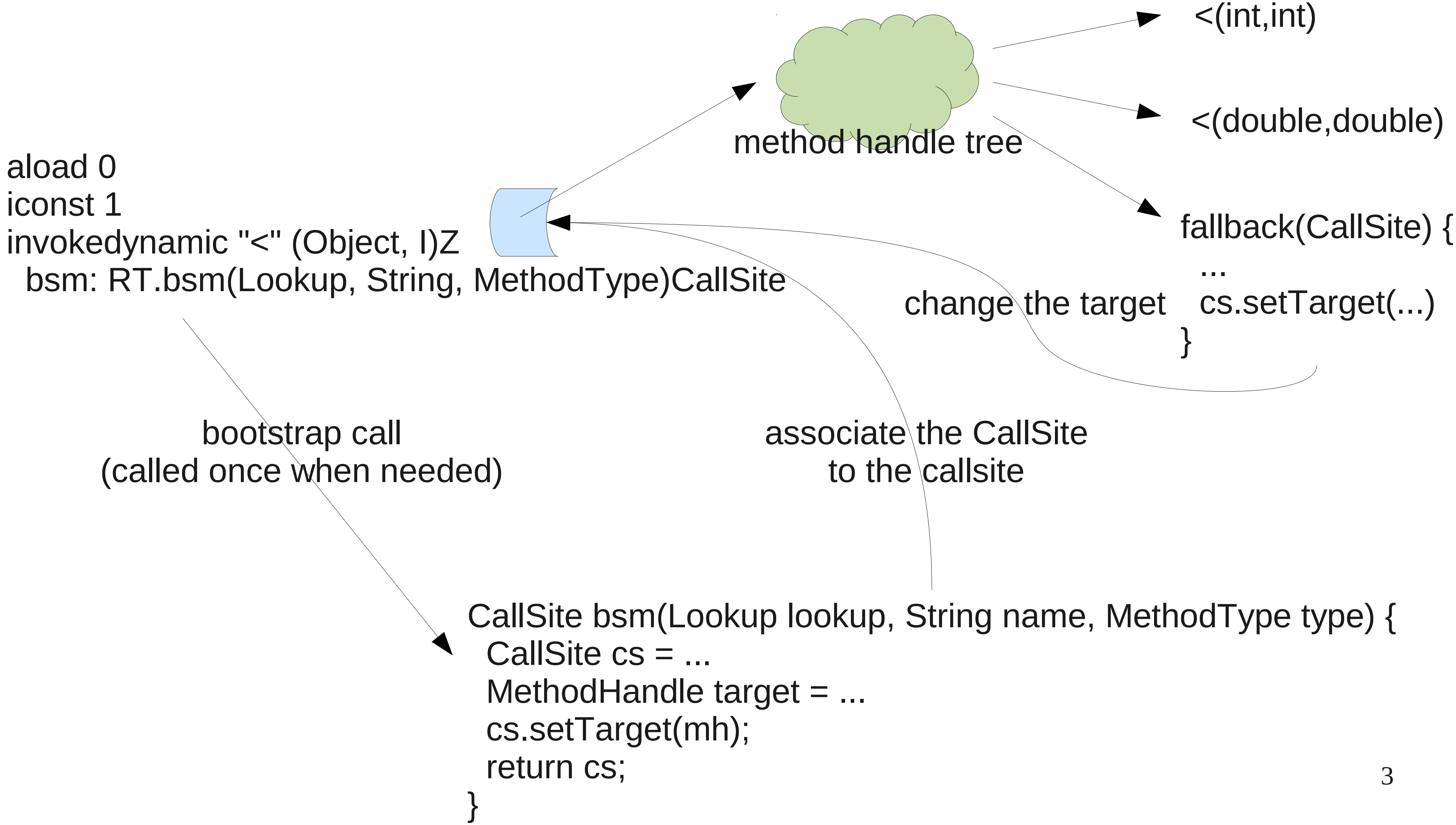
Goals

How to use JSR 292 to create common dynamic language runtime patterns

Try to gather best practices

Exercise the JSR 292 spec/implementations

invokedynamic



JSR 292

- enhanced bytecode + a new API
  - new constant pool constants
  - invokedynamic
    - link a callsite to one or several target method implementations ?
    - can relink dynamically !
  - java.lang.invoke
    - manage function pointers (MethodHandle)
    - combinators
    - provide adhoc classes: ClassValue, SwitchPoint

JSR 292 & Java

JSR 292 is **poorly supported** by Java (the language)

No support of

- invokedynamic
  - expando keyword ? => expando type, expando method
- constant method handle
  - Constant method reference (lambda) should be converted to a MethodHandle (not currently specified)

Cookbook

- interceptors
- constants lazy initialization
- callsite adaptation
    - varargs, spread, named parameters
- method dispatch
  - single-dispatch
    - monomorphic IC, unverified entry point, bi-morphic IC, dispatch table
  - double-dispatch
    - binary op, visitors
  - multi-dispatch
- metaclass & invalidation

Single Dispatch

The target method is chosen depending on the class of the receiver

```
array = [ 19, "baz", ... ]
foreach(array as value) {
    echo(value.to_str())
}
```

int::to_str()

string::to_str()

Foo::to_str()

... ::to_str()

Can't use a vtable !

Constructs a dispatch table ?

One by callsite or one by selector

Dispatch Table

- Hash map between a class and a method handle
- Use invoker + fold to insert the target in front of the arguments

```
DispatchMap dispatchMap = new DispatchMap() {
  protected MethodHandle findMethodHandle(Class<?> receiverClass) {
    MethodHandle target = ...
    return target.asType(type);
  }
};
MethodHandle lookupMH = MethodHandles.filterReturnValue(Object#getClass,
  DispatchMap#lookup.bindTo(dispatchMap));
lookupMH = lookupMH.asType(methodType(MethodHandle.class, type.parameterType(0)));
MethodHandle target = MethodHandles.foldArguments(
  MethodHandles.exactInvoker(type), lookupMH);
callsite.setTarget(target);

class DispatchMap {
  public MethodHandle lookup(Class<?> k) {
```

Perf ? How it works ?

- If a method handle is
  - Used in a hot code
  - Static or reachable from a static context
    - invokedynamic callsite
    - static final method handle
    - constant method handle (+ldc)
- The JIT will inline the whole method handle blob at callsite
  - Gold card: don't decrease inlining_depth
  - Others thresholds still exist (number of IR nodes, etc)

Perf ? Dispatch Table ?

So the method handles get from the dispatch table **aren't inlined** !

Use an inlining cache !

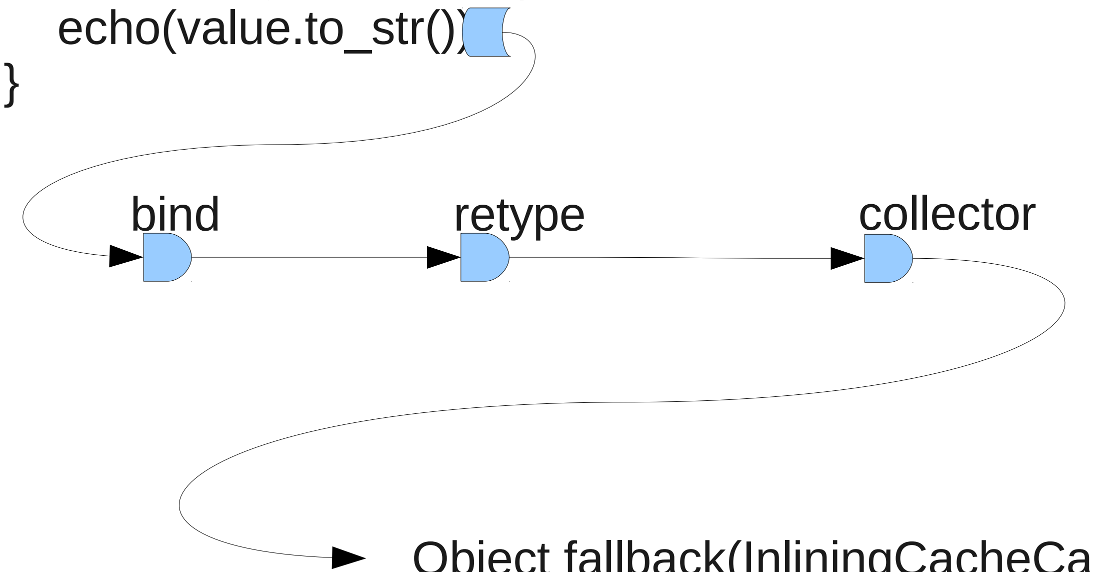
- Keep last returned MH and check if the receiver class has not changed

Constructs a tree of decision and fallback to a dispatch table if depth > threshold

Inlining cache

Delay the computation of the target until arguments are available

```
array = [ 19, "baz", 42.0 ]
foreach(array as value) {
    echo(value.to_str())
}
```



Object fallback(InliningCacheCallSite callSite, Object[] args) {

Inlining cache

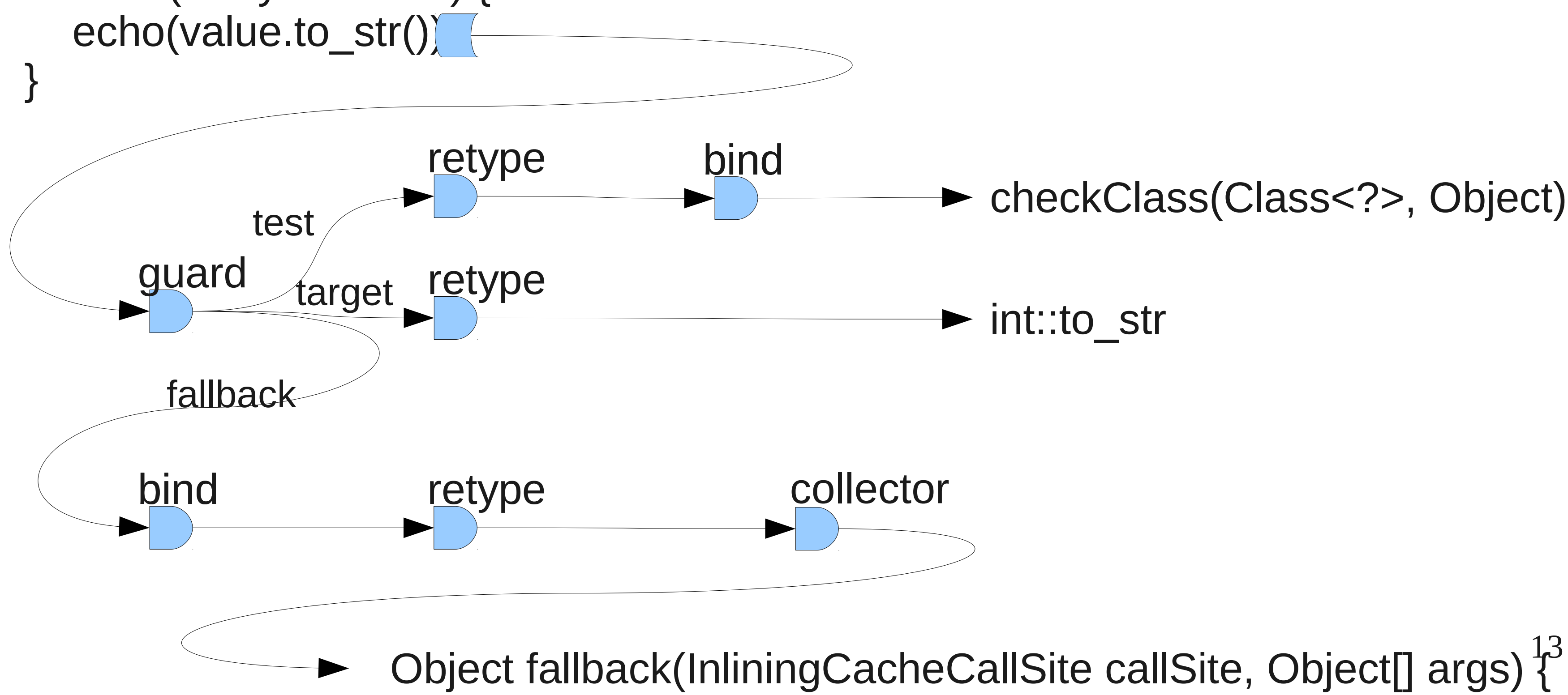
The BSM installs fallback as target

```
class InliningCacheCallSite extends MutableCallSite {
    ...
}

  static CallSite bootstrap(Lookup lookup, String name, MethodType type) {
    InliningCacheCallSite callSite = new InliningCacheCallSite(...);

    MethodHandle fallback = #fallback.bindTo(callSite);
    fallback = fallback.asCollector(Object[].class, type.parameterCount());
    fallback = fallback.asType(type);

    callSite.setTarget(fallback);
    return callSite;
  }

  static Object fallback(InliningCacheCallSite callSite, Object[] args) {
```

Inlining cache

Install a guard to avoid to do the lookup each time

```
array = [ 19, "baz", 42.0 ]
foreach(array as value) {
    echo(value.to_str())
}
```



Inlining cache

The guard fallback reuse the previous target

```
Object fallback(InliningCacheCallSite callSite, Object[] args) throws Throwable {
  MethodType type = callSite.type();
  Class<?> receiverClass = args[0].getClass();
  MethodHandle target = ...
  target = target.asType(type);
  MethodHandle test = #checkClass.bindTo(receiverClass);
  test = test.asType(test.type().changeParameterType(0, type.parameterType(0)));
  MethodHandle guard = MethodHandles.guardWithTest(test, target, callSite.getTarget());
  callSite.setTarget(guard);
  return target.invokeWithArguments(args);
}

static boolean checkClass(Class<?> clazz, Object receiver) {
    return receiver.getClass() == clazz;
}
```

Inlining cache

A chain of guards too big will kill performance

Store the depth in the CallSite object

```
Object fallback(InliningCacheCallSite callSite, Object[] args) throws Throwable {
  MethodType type = callSite.type();
  if (callSite.depth >= MAX_DEPTH) {
    ...
  }
  Class<?> receiverClass = args[0].getClass();
  MethodHandle target = ...
  target = target.asType(type);
  MethodHandle test = #checkClass.bindTo(receiverClass);
  test = test.asType(test.type().changeParameterType(0, type.parameterType(0)));
  MethodHandle guard = MethodHandles.guardWithTest(test, target, callSite.getTarget());
  callSite.depth++;
  callSite.setTarget(guard);
  return target.invokeWithArguments(args);
}
```

Inlining cache – Thread safety

- The code is not thread safe
  - race to call setTarget
    - no problem it's a cache
  - depth not atomic !
    - real depth may be greater than MAX_DEPTH
      - in slow path, so better to use volatile + CAS (AtomicInteger)

```
Object fallback(InliningCacheCallSite callSite, Object[] args) throws Throwable {
  MethodType type = callSite.type();
  if (callSite.depth.get() >= MAX_DEPTH) {
    ...
  }
  ...
  callSite.depth.incrementAndGet();
  callSite.setTarget(guard);
  return target.invokeWithArguments(args);
}
```

How to improve the dispatch table ?

Solution if few method handles:

- Dispatch map (class -> int) + **Switch inliner**
  - switchInliner(MH fallback, MH... mhs)
    - All method handles must have the same signature
    - Returns a mh that takes a supplementary first parameter (the index) and calls mhs[index].

      The JIT should try to inline the all mhs
- Also solve problem of several class that share the same method implementation (inheritance)

Binary operations

Double dispatch, the target method depends on the class of the two arguments

```
i = ...
j = 0
while(true) {
   i, j = i + j, i
}
```

+(int, int): Object (int or BigInteger)

+(BigInteger, int): BigInteger

+(BigInteger, BigInteger): BigInteger

Moreover, int operations can overflow to BigInteger

```
Object fallbackOpBoth(BinOpCallsite callSite, Object value1, Object value2) {
  Class<?> class1 = value1.getClass(), class2 = value2.getClass();
  MethodHandle target, guard1, guard2;
  if (class1 == BigInteger.class) {
    guard1 = BIGINTEGER_CHECK;
    if (class2 == BigInteger.class) {
      guard2 = BIGINTEGER_CHECK2;
      target = BigInteger#add;
    } else {
      if (class2 != Integer.class) { throw ... }
      guard2 = INTEGER_CHECK2;
      target = MethodHandles.filterArguments(
                  BigInteger#add, 1, OBJECT_TO_INTEGER_TO_BIGINTEGER);
    }
  } else {
    ...
  }
  target = target.asType(callSite.type());
  MethodHandle fallback = callSite.getTarget();
  MethodHandle guard = MethodHandles.guardWithTest(guard1,
      MethodHandles.guardWithTest(guard2, target, fallback),
      fallback);
  callSite.setTarget(guard);
  return target.invoke(value1, value2);
}
```

Perf warning: unboxing trouble !

Conversion Object -> int is not equivalent to Object -> Integer -> int

```
mh = BigInteger#valueOf(long);
mh = mh.asType(methodType(BigInteger.class, int.class));
mh = mh.asType(methodType(BigInteger.class, Integer.class));
mh = mh.asType(methodType(BigInteger.class, Object.class));
OBJECT_TO_INTEGER_TO_BIGINTEGER = mh
```

Object -> int accepts Byte -> byte -> int

Binary operations

Signature is fixed, 2 parameters, fixed small number of classes

- => no dispatch table needed

A lot of operation involve one constant

- x + 1, 1 + x, etc
- => no need to do a double dispatch

Usually overflow are rare

- => no need to construct the whole tree of possibility
- If one arg is constant
  - => can use dedicated overflow test !

Overflow test for +

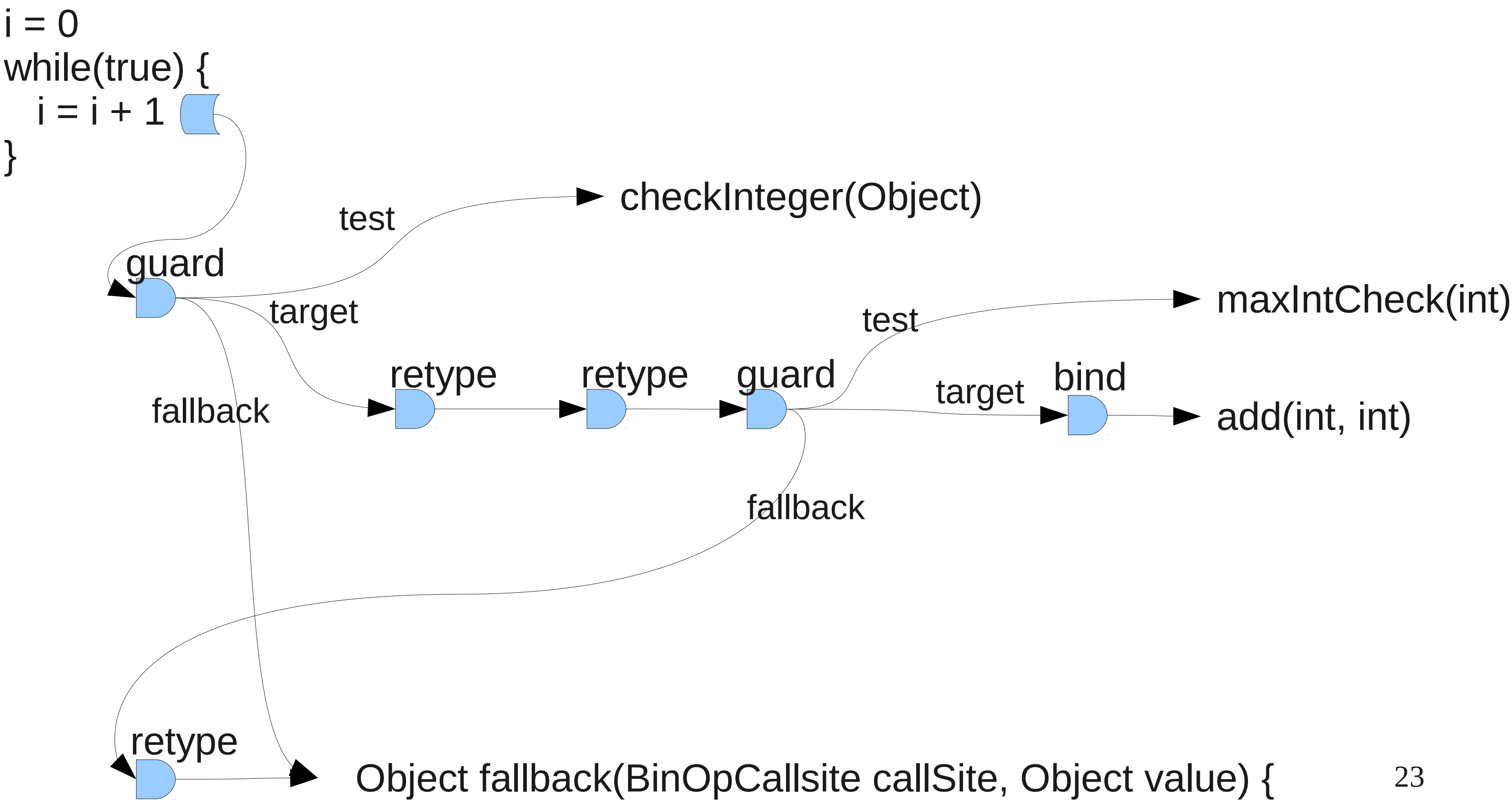
Check if values as same sign and result an opposite sign

```
Object safeAdd(int value1, int value2) {
    int result = value1 + value2;
    if ((value1 ^ result) < 0 && (value1 ^ value2) >= 0) {
      return BigInteger.valueOf(value1).add(BigInteger.valueOf(value2));
    }
    return result;
  }
```

It can be simplified, the overflow test for x + 1 is
  if (x == Integer.MAX_VALUE)

Overflow for x + cst

Reuse the same fallback for the guards



Overflow for x + cst

invokedynamic with one argument,
constant value is sent as a bootstrap argument

```
Object fallbackOpLeft(BinOpCallsite callSite, Object value) {
  ...
  MethodHandle op = MethodHandles.insertArgument(#add(int, int), 1, callSite.value);
  MethodHandle overflowTest = createAddOverflowTest(rightValue);
  if (overflowTest != null) {
    overflowGuard = MethodHandles.guardWithTest(overflowTest,
        op,
        fallback.asType(methodType(Object.class, int.class)));
  } else {
    overflowGuard = op;
  }
  overflowGuard = overflowGuard.asType(methodType(Object.class, Integer.class)).
                                asType(methodType(Object.class, Object.class));
  MethodHandle target = MethodHandles.guardWithTest(INTEGER_CHECK,
     overflowGuard, fallback);
```

VM Optimization ?

x *binop* cst optimization enables the VM to do further optimizations

```
i = 0
while(i<array.length) {
   echo(array[i])
   i = i + 1
}
```

The VM should fold the 3 Integer checks
then range check optimization can be applied
then escape analysis remove boxing
=> same loop as Java :)

Metaclass & invalidation

The metaclass provides language specific class metadata

Metaclass can be mutable

```
MethodHandle mh = lookup.findVirtual(String.class, "toLowerCase",
    methodType(String.class));
for(int i=0; i<10; i++) {
  System.out.println("Hello".toUpperCase());  // invokedynamic
  if (i == 3) {
    MetaClass.getMetaClass(String.class).
        redirect("toUpperCase", methodType(String.class), mh);
  }
}
```

ClassValue

Association between a class object and a runtime specific object
Act has a concurrent weak hash map

```
public class MetaClass {
  private final HashMap<Selector, MethodHandle> vtable =
    new HashMap<Selector, MethodHandle>();

  private static final ClassValue<MetaClass> metaClassValue =
    new ClassValue<MetaClass>() {
    protected MetaClass computeValue(Class<?> type) {
      return new MetaClass(...);
    }
  };

  public static MetaClass getMetaClass(Class<?> clazz) {
    return metaClassValue.get(clazz);
  }
```

Managing mutation

- Pull
  - Volatile serial number in MetaClass
    - Bind the serial current value in the method handle blob
    - Bind the metaclass + field accessor
    - Increment when metaclass is mutated
- Push
  - 1 frozen MetaClass <--> 1 SwitchPoint
    - Bind the SwitchPoint
      - Volatile boolean when interpreted
      - No check when JITed + dependencies for invalidation
    - Create a new SwitchPoint if mutated

SwitchPoint

Insert a SwithPoint check after the class check and before the target adaptation

```
Object staticFallback(InvokeStaticCallSite cs) {
  MetaClass metaClass = MetaClass.getMetaClass(cs.ownerType);
  MethodType type = cs.type();
  MethodHandle mh;
  SwitchPoint switchPoint;
  synchronized(MetaClass.MUTATION_LOCK) {
    mh = metaClass.staticLookup(cs.name, type);
    switchPoint = metaClass.switchPoint;
  }
  if (mh == null) { mh = ... }
  mh = mh.asType(type);
  MethodHandle target = switchPoint.guardWithTest(mh, fallback);
  ...
```

SwitchPoint in HotSpot

SwitchPoint is currently implemented with a volatile check but

Pull then push optimization:

- Detects that SwitchPoint is hot (specific profiling)
- Make it a no-op + store callsites using a switch point in the switchPoint + de-optimization if invalidation

=> as fast as an inlining cache call !

Invalidation & inheritance

Need to maintain a metaclass hierarchy to also invalidate sub-metaclasses

```
private static final ClassValue<MetaClass> metaClassValue =
  new ClassValue<MetaClass>() {
  protected MetaClass computeValue(Class<?> type) {
    Class<?> superclass = type.getSuperclass();
    MetaClass parentMetaClass = (superclass == null)? null: getMetaClass(superclass);
    return new MetaClass(parentMetaClass);
  }
};
private final LinkedList<WeakReference<MetaClass>> subMetaClasses =
  new LinkedList<>();
MetaClass(MetaClass parent) {
  synchronized(MUTATION_LOCK) {
    switchPoint = new SwitchPoint();
    this.parent = parent;
    if (parent != null) {
      parent.subMetaClasses.add(new WeakReference<MetaClass>(this));
    }
} }
```

Bulk invalidation

To try to avoid deoptimization flood

```
public void redirect(String name, MethodType type, MethodHandle target) {
    synchronized(MUTATION_LOCK) {
      ArrayList<SwitchPoint> switchPoints = new ArrayList<>();
      mutateSwitchPoints(this, switchPoints);
      SwitchPoint.invalidateAll(switchPoints.toArray(new SwitchPoint[switchPoints.size()]));
      vtable.put(new Selector(name, type), target);
    }
  }
  private static void mutateSwitchPoints(MetaClass mc, List<SwitchPoint> switchPoints) {
    switchPoints.add(mc.switchPoint);
    mc.switchPoint = new SwitchPoint();
    for(Iterator<WeakReference<MetaClass>> it = mc.subMetaClasses.iterator(); it.hasNext();) {
      MetaClass subMC = it.next().get();
      if (subMC == null) { it.remove(); continue; }
      mutateSwitchPoints(subMC, switchPoints);
    }
  }
```

Metaclass pattern & prototype

- This pattern doesn't work with prototype based (not class based) languages
  - Self, JavaScript, Seph, etc.
- Solutions ?
  - Use pseudo class trick (V8)
    - JVM will require two guards instead of 1
      - All Objects (even String) implement get/setMetaClass()
        - interface injection?
  - Use profiling + allocation site class

Overall recommendations

- Compiler should have a type inference pass
  - and check dead-code/missing return
- Segregate fast path/slow path
- No second class citizen
  - No RubyObject, GroovyObject, etc.
- Design with concurrency in mind

Questions ?

https://code.google.com/p/jsr292-cookbook/