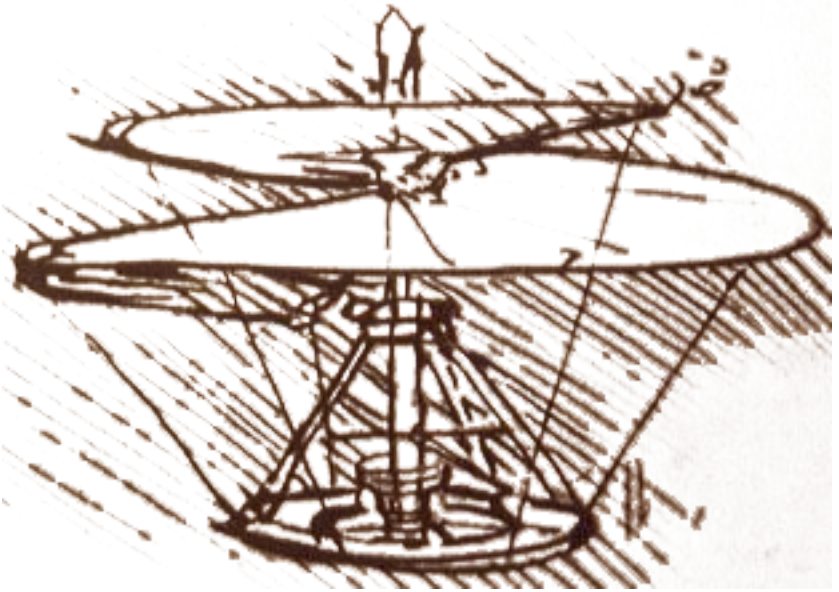
JSR 292 and companions

Rémi Forax



FOSDEM'09

Da Vinci Mission

- Prototype (J)VM extensions to run non-Java languages efficiently
- Complete the existing architecture with general purpose extensions
- New languages to co-exist gracefully with Java in the VM
- First-class architectural support (not hacks or side-cars)

VM Extensions

- Dynamic invocation
- Lightweight method objects
- Interface Injection
- Lightweight bytecode loading
- Tail calls and tail recursion
- Continuations and stack introspection
- Tuples, value type, immediate wrapper types
- ...

First round : JSR 292

- Dynamic invocation
- Lightweight method objects
- Interface Injection
- Lightweight bytecode loading
- Tail calls and tail recursion
- Continuations and stack introspection
- Tuples, value type, immediate wrapper types
- ...

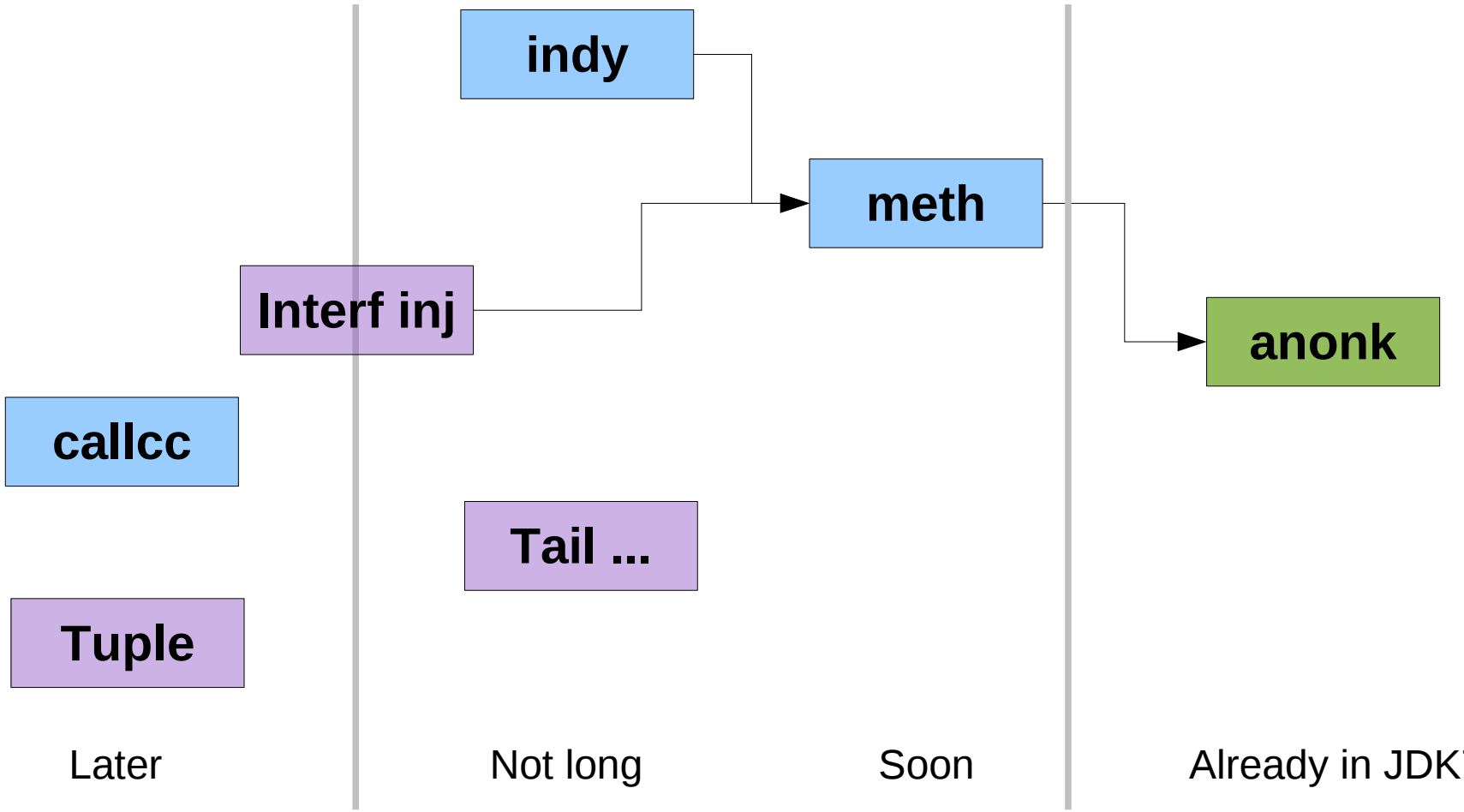
JSR 292

JSR 292 RI

MLVM patch queue

- Dynamic invocation (indy)
- Lightweight method objects (meth)
- Interface Injection soon !
- Lightweight bytecode loading (anonk)
- Tail calls and tail recursion soon !
- Continuations and stack introspection (callcc)

Patch dependency graph and integration



Dynamic Invocation (indy)

- Invokedynamic: a new bytecode instruction
  - Let language developers implement their own method selection
    - static part: bootstrap method (Java code)
    - dynamic part: ability to create decision tree
- Need to surface (safe) function pointers
  - Lightweight method objects:
    - java.dyn.MethodHandle

invokedynamic

Format

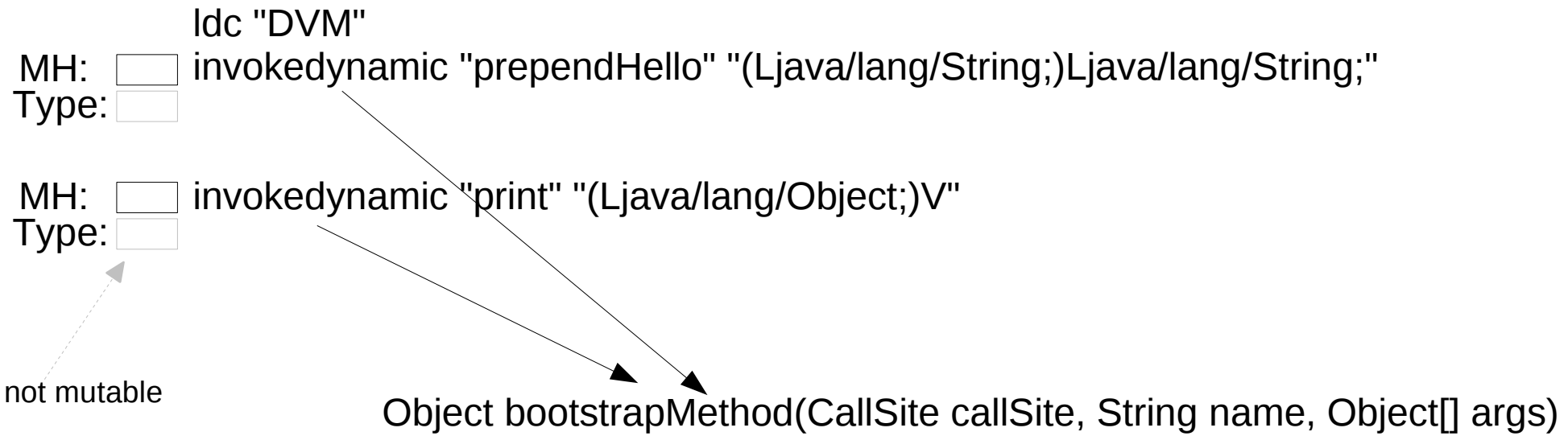
invokedynamic
indexbyte1
indexbyte2
0
0

Forms *invokedynamic* = 186 (0xba)

- No receiver type (a real function call)
- 5 bytes:
  - byte1 and byte2 encode a CONSTANT_NameAndType
  - Two empty slots:
    - Hotspot stores a reference to oop in constant pool (CallSite)

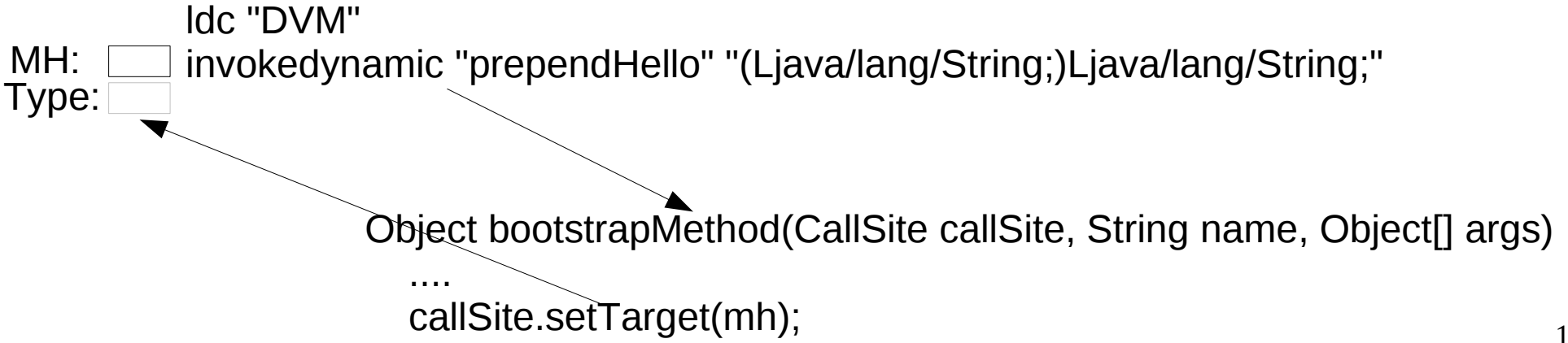
Dynamic Invocation

- At call site
  - If no registered MH,
    call a special bootstrap method (argument boxing)
  - else call MH (no boxing)



Bootstrap method

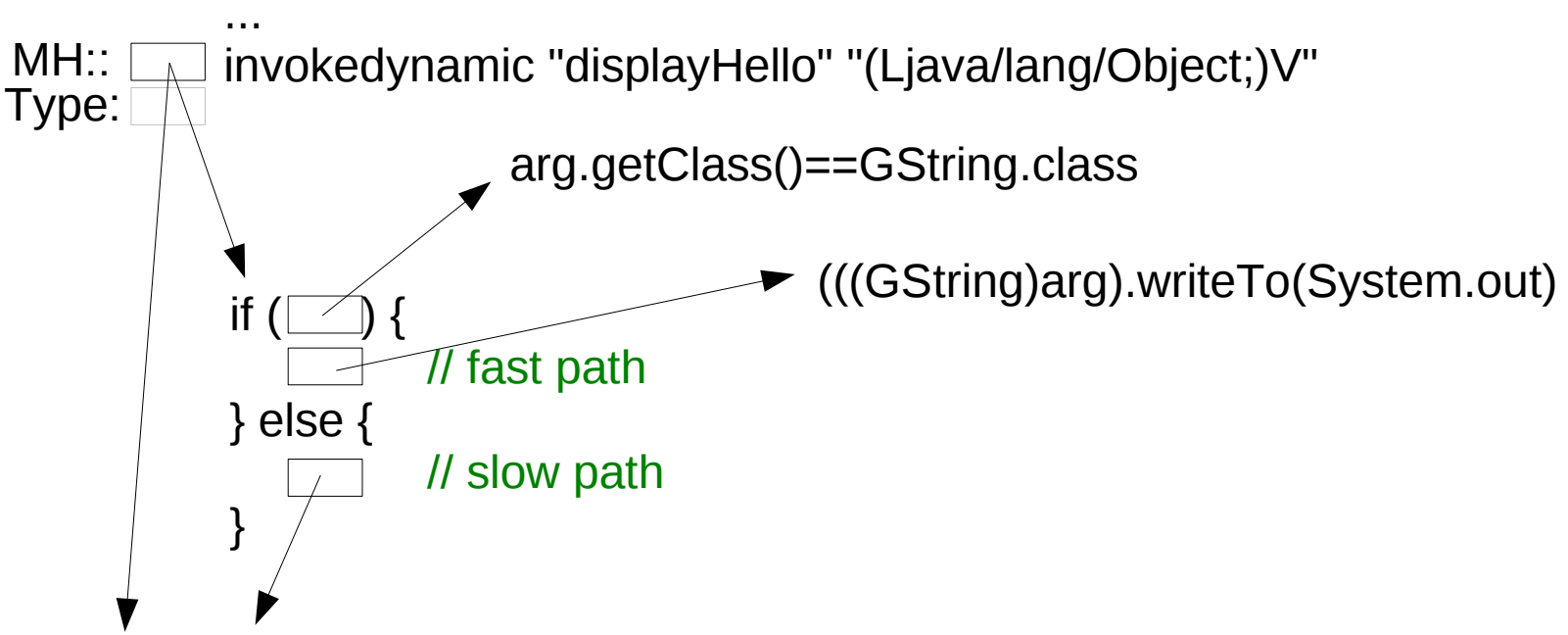
- Returns a value, may install a MH
- setTarget() check its type against call site type (using ==)
  - CallSite: (method handle + method type)
  - MethodHandle: (VM magic oop + method type)



Creating a MethodHandle (meth)

- java.dyn.MethodHandles
  - findVirtual, findSpecial, etc.
  - Adapting an existing method handle
    - Add/remove an object to/from the arguments
    - Convert arguments (primitive conv., checkcast, boxing)
    - Spread/collect array (varargs)
  - Composing existing method handles
    - If: guardWithTest()
    - Composition: makeDispatcher()

An example of decision tree




```
slowPath(CallSite callSite, Object arg) {
  callSite.setTarget(MethodHandles.guardsWithTest(
     MOP.find("checkClass", arg, arg.getClass()),
     MethodHandles.convertArguments(
        MOP.find("displayHello", arg),
        mtype(" + descName(arg)+ ")V")
     ),
     callSite.getTarget())
  );
}
```

MethodHandle in hotspot

- 4 kinds, 2 big strategies :
  - Direct, Virtual, Bound:
    - Store directly a vmEntry/methodOop
  - Adapter:
    - If no allocation: use VM call stubs
    - Else: auto-generate a special method "invoke" and create a methodOop on it

Invoking a MethodHandle

- Using invokevirtual directly on a method handle

```
static void call(MethodHandle mh) {
  ldc "Hello DVM"
  aload 0
  invokevirtual "java/dyn/MethodHandle" "invoke" "(Ljava/lang/String;)V"
  ...
```

- At invocation time,
  check MH type against descriptor type
- In Hotspot, MethodHandle class has a special layout with an extra vtable populated lazily

Anonymous Klass (anonk)

- Can create a class (from a byte array)not loaded by a classloader
- Take a host class to get security context
- Allow to patch the constant pool before its definition

- Allow to create small macro stubs
  => adapter method handle

Interface Injection (work in progress)

- Interface must to be marked injectable for performance reason
- If invokeinterface/instanceof(*)/checkcast fails, upcall to a Java code
- Each type can be injected once, supertype first
- For concrete type, Java code should provide MethodHandles for all method of the interface that aren't already defined

* to produce a result

JSR 292 Backport

- In few words :
  - Use ASM to transform bytecode at compile time or load time
    - ASM must be able to read invokedynamic bytecode
  - MethodHandle are interfaces, like BGGA closures but generated at runtime
  - Adapters are chained bytecode weavers that are able to generate one method for all chained adapters
  - Indy part is not implemented (will be !)

Questions ?