JSR 292 on Android

Jerome Pilliet / Remi Forax
University Paris East

JVM Summit'13

Us

Rémi Forax,
Teaching Assistant
Expert for JSR 292 & JSR 335

Jérôme Pilliet,
Phd Student,
Android Hacker

Why ?

- We like dynamic language runtime devs
  - No need to keep two versions of a runtime (pre-invokedynamic/post-invokedynamic)
- We like dynamic language users
  - Some languages only run on the JVM
- We like Java
  - Java 8 (lambda) uses invokedynamic

but first a demo !

BTW, we're lying

We don't implement JSR 292 on Android

because Dalvik is not a Java VM

We first need to adapt the JSR 292 to Android

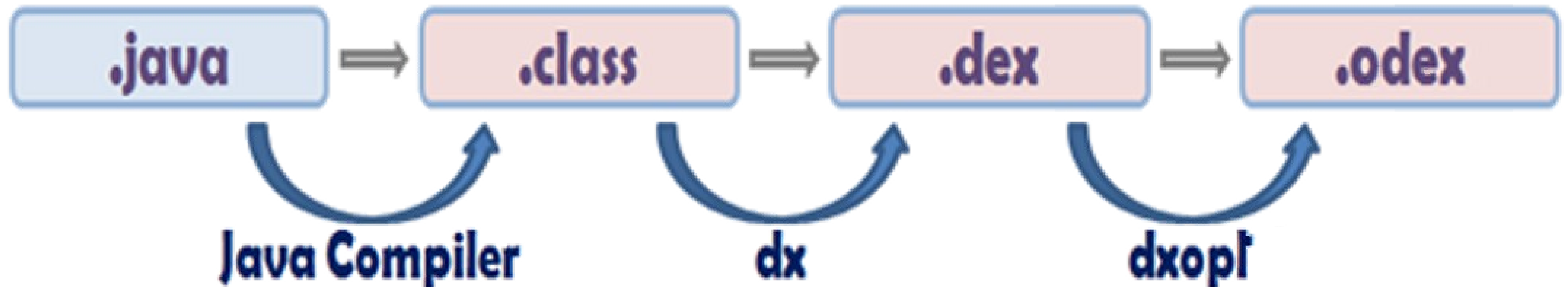
- Dalvik opcodes are register based
- Better spec (obviously!)

and then implement it

- Smartphones/Tablets are constraint environments

A Glimpse to Android

Toolchain



The DEX file

- is read-only (mapped)
- contains several classes (more like a jar)
- one constant pool by type (String, Prototype, ...)
- optimized ODEX
    - verified offline (at install time)

Dalvik

- Register based opcodes
- Several interpreters
  - **Portable,** armv***,** x86
- A 'java' dev kit (libcore) written from scratch
- An incremental non-generational GC
- A very small JIT (for benchmark)

JSR 292 implementation for Android

A new DEX format

- A new header
- 5 new instructions
  - invoke-exact/invoke-generic/invoke-dynamic
  - const-methodtype/const-methodhandle
- 4 new constant pools
  - methodType/methodHandle/invokeDynamic/bsmArgs

new invoke opcodes

- invoke-exact/range: 40/42
  invoke-generic/range: 41/43
  - opcode {vC, vD, vE, vF, vG} mt@BBBB
  - opcode {vCCCC...vNNNN} mt@BBBB

- invoke-dynamic/range: 73/79
  - opcode {vC, vD, vE, vF, vG} indy@BBBB #ZZZZZZZZ
  - opcode {vCCCC...vNNNN} indy@BBBB #ZZZZZZZZ

MethodType

- Interned in a concurrent weak hashmap in Java
- Opaque pointer in C

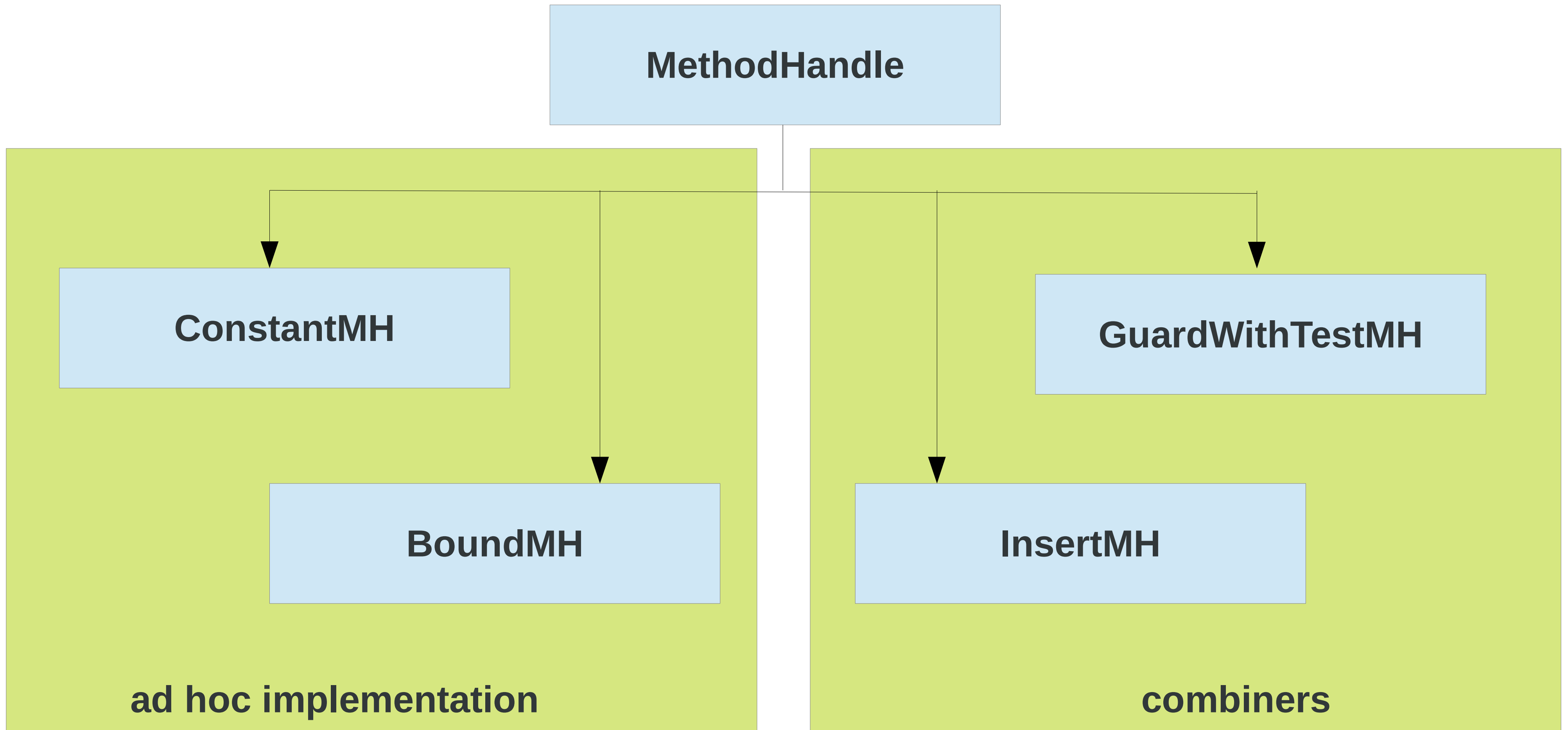
- invoke-exact/invoke-generic have an offset in a table of resolved MethodType
- Invoke-dynamic and const-methodType doesn't intern it

MethodHandle

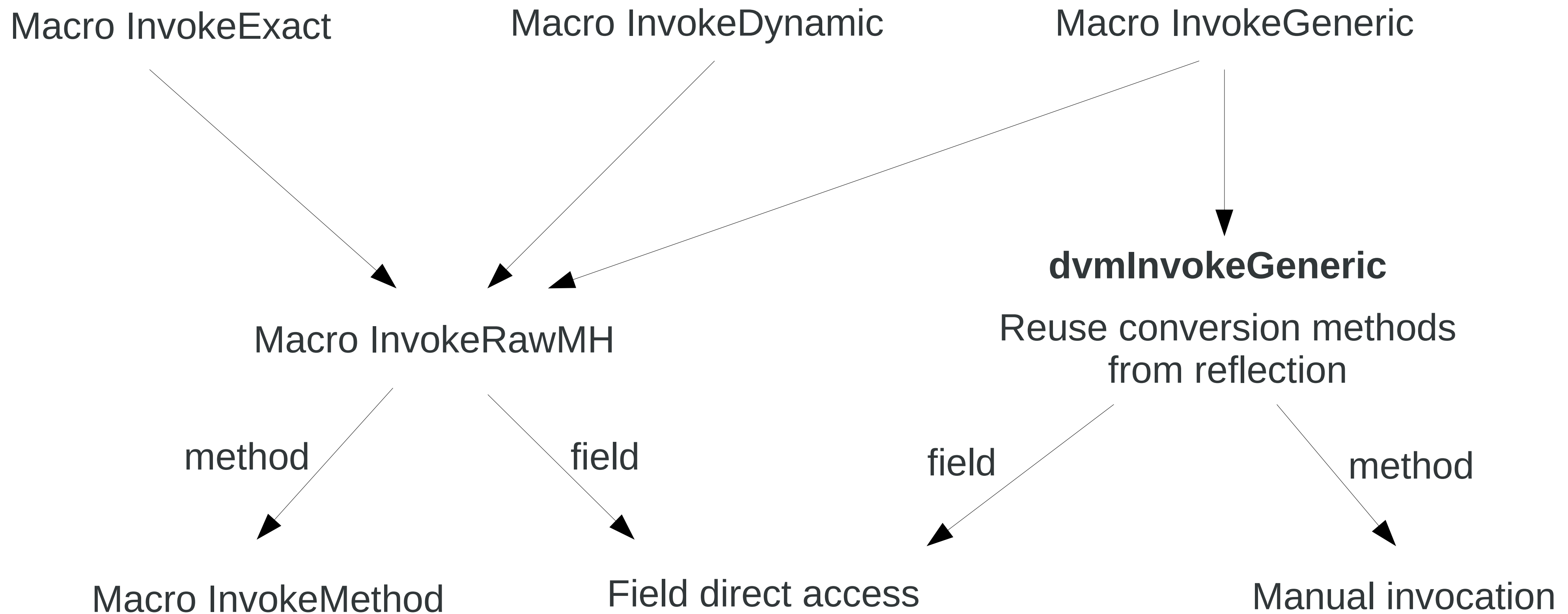
- 3 rings of MethodHandle
  - Constant method handle
  - Method Handle for lambdas (bind/constant)
  - Method Handle combiners

- In memory
  - kind (field:get/set*instance/static*volatile +
    method:static/direct/super/virtual/interface/newInstance +
    bind + constant +
    AST-interpreter + mini-interpreter (combiner))
  - slot (field index, directMethod index, vtable/itable index)
  - declaringClass
  - methodType

MethodHandle hierarchy

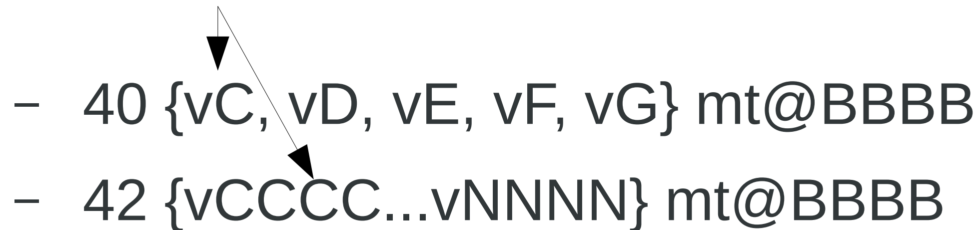


Macros and functions in the interpreter



InvokeExact

method handle



- 40 {vC, vD, vE, vF, vG} mt@BBBB
- 42 {vCCCC...vNNNN} mt@BBBB

Implementation scketch:

get methodHandle (vC)

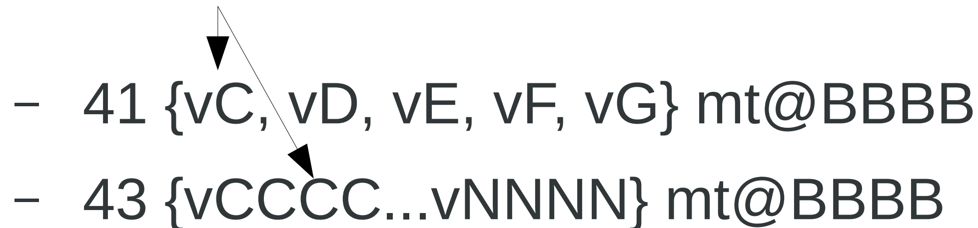
rebuild arguments (drop vC)

get methodType (mt@BBBB)
get methodType from methodHandle
check method types
if false then throw WrongMethodTypeException !

call the macro InvokeRawMH

InvokeGeneric

method handle



- 41 {vC, vD, vE, vF, vG} mt@BBBB
- 43 {vCCCC...vNNNN} mt@BBBB

Implementation scketch:

get methodHandle (vC)

rebuild arguments (drop vC)

get methodType (mt@BBBB)
get methodType from methodHandle
check method types
if true then like invokeExact !
else then

convert arguments (boxing, unboxing, cast, …)

create a new stack frame and push arguments

InvokeDynamic

- 73 {vC, vD, vE, vF, vG} indy@BBBB #ZZZZZZZZ
- 79 {vCCCC...vNNNN} indy@BBBB #ZZZZZZZZ

Implementation scketch:

get the current class

get call site index (#ZZZZZZZZ)

get call site (currentClass→callSites[callsite_idx])

if NULL then

get indy reference (indy@BBBB)

get method handle, method type and bsm arguments (indy)

java call: call_bootstrap(mh, mt, bsm_args)

in java: create lookup and invoke-exact? the bootstrap method

intern the call site (CAS)

call the macro InvokeRawMH (callSite→target) // a volatile read !

Garbage Collector interaction

- Some opcodes use Java references
  - invoke-dynamic access to a CallSite
  - invoke-exact/generic access to a MethodType
- MethodTypes for invoke-exact/generic are interned, stored as strong reference in the global hashtable (in Java)
- CallSites are stored in ClassObject (VM) and are roots of the GC marking phase

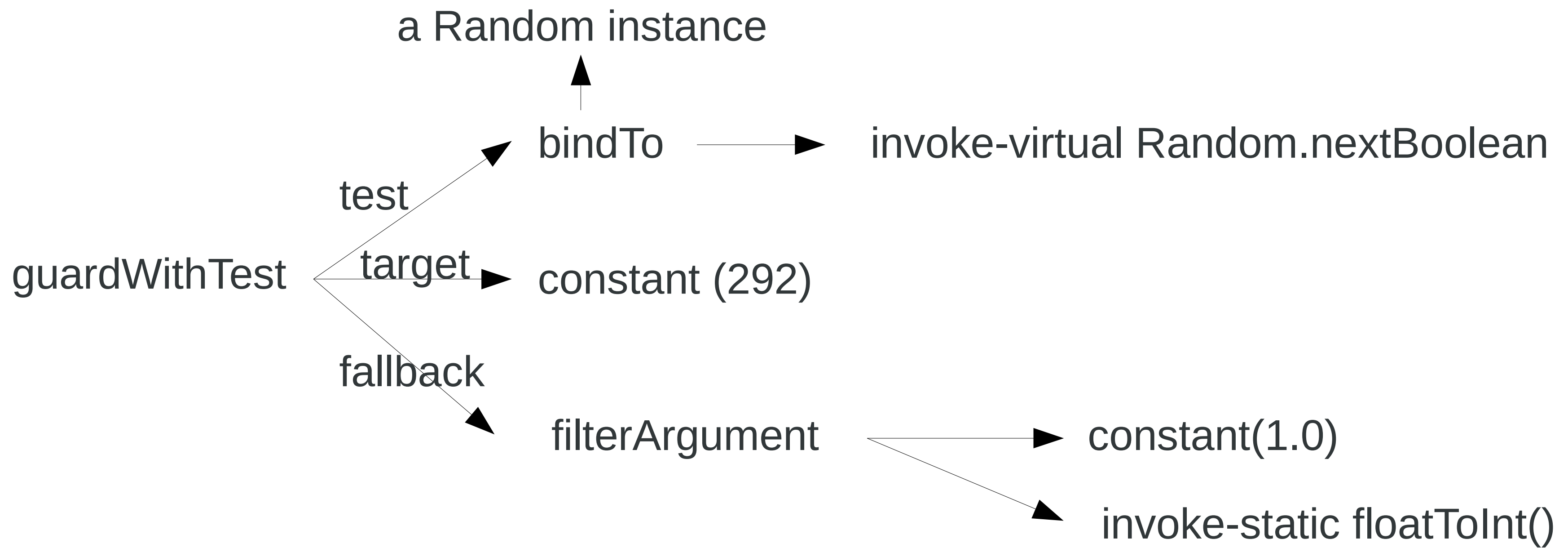
MethodHandle combiners

- Combiners are small reusable bricks, language runtimes tend to use a bunch of them
- Some method handles are called once, some are called in hot-loop

- Two implementations
  - In Java (AST interpreter)
  - In C, with a specific stack frame (mini-interpreter)

Example of combiners

(random.nextBoolean())? 292: (int) 1.0;



AST interpreter (Java)

- Special method handle kind(AST)
- Upcall to a non-public Java method, box all arguments, unbox return value

```
class MethodHandle {
  /*package*/ Object invokeAST(Object[] args) {
    ...
 } }
```

AST interpreter (Java)

```
class GWTMH extends MethodHandle {
  private final MethodHandle test;
  private final MethodHandle target;
  private final MethodHandle fallback;

  @Override
  /*package*/ Object invokeAST(Object[] args) {
    if ((Boolean)test.invokeAST(args)) {
      return target.invokeAST(args);
    }
    return fallback.invokeAST(args);
  } }
```

```
class MethodHandle {
  /*package*/ Object invokeAST(Object[] args) {
    switch(args.length) {
      case 0: return invoke();
      case 1: return invoke(arg[0]);
      ...
    } } }
```

Mini-interpreter

- Generate a specific code corresponding to the whole method handle tree

  (by walking the AST)

- Do the execution in one stack frame

  The whole stack frame needs to be fully typed (at generation time)

  => use index indirection

- A specific interpreter inside the interpreter ?

Mini-interpreter example

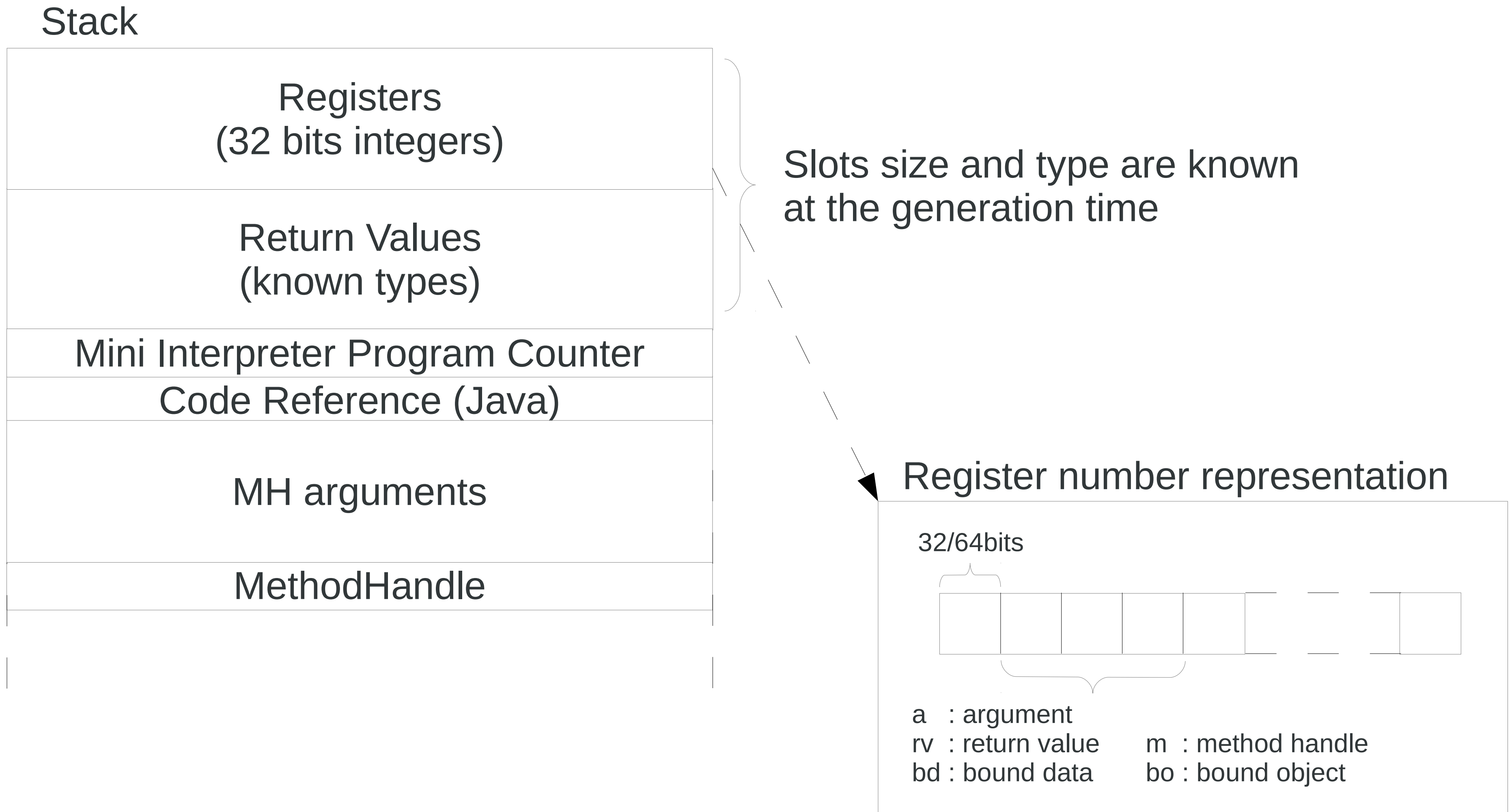
```
gwt = MHs.guardWithTest(test1, target, fb1);
  ↳ test1 = MHs.insertArgument(test0, 0, new Random());
      ↳ test0 = MHs.lookup().findVirtual(Random.class, “nextBoolean”, MT(boolean.class));
  ↳ target = MHs.constant(int.class, 292);
  ↳ fb1 = MHs.filterArguments(fb0, 0, f2i);
      ↳ fb0 = MHs.constant(double.class, 1.0);
      ↳ f2i = MHs.lookup().findStatic(Convertions.class, “floatToInt”, MT(int.class, float.class);
```

r	rv1/32 or bd2		0
rv	1	int	1
	true or false	boolean	0
	code + 8		
	0x...		
a			0
	MethodHandle		

```
gwt.nb_r = 1        gwt.nb_rv = 2
gwt.mhs = { test1, test0, target, fb0, f2i  }

code {
  const r0 bd0                          // ldc Random object
  invoke-frame-mh rv0 m1 r0 1   // call test0
  ifne rv0 :else
  const r0 bd2                                  // ldc 292
  goto :end
  :else
  const r0 bd3                                  // ldc 1.0
  invoke-frame-mh rv1 m4 r0 1   // call f2i
  const r0 rv1
  :end
  invoke-frame-mh 0 null r0 1     // return
}
```

Mini-interpreter Stack Frame



Mini-Interpreter Twist

Davik is already a register based VM

So we can re-use the same opcodes

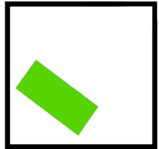
So we can re-use the same interpreter

- We need one supplementary opcode invoke-frame-mh

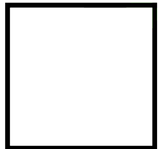
Where we are ?

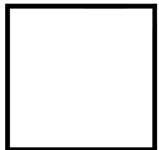
Android SDK compile with java 7

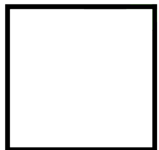
Modify dx/dexopt/vm for the new class format


Implement direct method handle in the interpreter

invoke-generic is not finished !


Combiner and AST interpreter


Mini interpreter


Lambda ?

Question ?

https://bitbucket.org/jpilliet/android-sdk-292