Retrospect Implementation Notes

HOLOGRAMS

* Class rewriting
  + JVM doesn’t allow manually defining class in java.\* package, so have to rename at least some classes
    - In JDI implementation, can rename back to hide this as an implementation detail
* Code size
  + Need to guarantee code doesn’t get too much bigger!
* Object initialization
  + Native objects
    - Generate no-argument constructor
    - Technically works – VM doesn’t verify final fields are assigned
    - Not ideal because VM might be free to cache values, ignore assignments, etc.
    - Could also use FieldMapMirrors instead – may be faster than reflection-based NativeObjectMirrors
    - Should refactor mirror interface to use constant pool offsets instead of field names
      * Can dynamically generate native mirrors that use a switch to select the right field
    - Note: two different kinds of “native” objects
      * Instantiated inside MirageClassLoader classes – all fields lifted to mirages already
      * Instantiated outside – all fields have to be lifted as they are accessed
    - Direct datatype implementation may still be fastest since all fields must be of type ObjectMirror?
      * Maybe not, can still separate primitives from references
  + Mirage objects
    - Add extra mirror field to constructors, pass up super chain
      * Doesn’t handle reflection (i.e. Class#newInstance)
      * In fact, messes up reflection entirely since the parameter types are wrong
        + Need to advice reflection calls to hide changes in general – aligns with principle of mirror-based architectures
      * How to distinguish calls to new (where mirror should be instantiated) from calls to super (where mirror should be passed along?)
        + Both are just INVOKESPECIAL
        + Do dataflow analysis that respects difference between uninitialized and uninitialized this
        + Mirror instantiated on caller side
      * Reflective construction (where mirror is already available) uses constructor that takes an Object
        + Must have different signature to distinguish from when the original class has a nullary constructor, to which we add the extra mirror argument
    - Instead, two ObjectMirage constructors (NO)
      * Nullary for new statements – creates native mirror inline
      * One that takes a mirror for calls to ObjectMirage.make
      * Problem
        + In Java, anonymous inner class constructors will set fields before calling super
        + Bytecode in general can do this whenever it wants
* Static initializers
  + Removed from classes already present in heap dump, since they were already initialized
  + For classes newly defined by the holograph VM (new bootstrap classes or those defined by defineClass, forced to execute at the right time via setting a dummy static field on the mirage class
* Static fields
  + ClassMirror interface for getting fields
  + Single ClassMirrorLoader instance per ClassLoader
  + Problem – can’t ClassLoader.loadClass() on non-public classes, even though it’s easy for many classes to get a hold of the Class instance by calling getClass() on an instance
  + No longer a problem since not mixing “native” implementations with others any more
* ClassLoading
  + MirageClassLoader
    - Give it two class loaders for delegation
      * One for mirror implementations
      * One for loading original code (for transforming to mirage classes)
    - Desirable to catch setup errors where bytecode provided by class loader doesn’t match original bytecode
      * Any missing/incorrectly typed fields
      * Any incorrect *serialVersionUID* fields?
  + Mirrors turn out to be useful for bytecode verification/analysis
    - Type checking and analysis of mirror tree instead of real classes avoids class circularity errors
    - After realizing I need to actually run ClassLoader.loadClass, this doesn’t work so well (yet?) because resolving links between ClassMirrors requires it
  + ClassLoaderMirror
    - loadClassMirror() should only find classes that were already loaded
      * Essentially implementation of native method ClassLoader#findLoadedClass0()?
    - Would need to actually execute mirage code to implement full loadClass() logic…
      * And I do now!
    - Should be called ClassMirrorLoader, is actually an object now
    - No more cheating with mirrors that see code from different VMs, VM now totally sandboxed just like you were debugging a remote process
  + Loading new classes
    - ClassLoader.defineClass one of the supported native methods
      * Needed for some very core reflection methods
  + Finding bytecode
    - Easy for holograph use to hit classes that weren’t (yet) defined in the original VM
      * Need to map name to bytecode
    - Finding bytecode missing from heap dump
      * Similar problem to the above
    - Initially attempted providing 1-to-1 map from ClassMirrorLoaders to live ClassLoader instances, but had problems
      * Tricky to identify instances, need app-specific rules like OSGi bundle name and version
      * Live ClassLoaders may not be able to actually load and define old classes, or even find their \*.class file resources, without conflicts or runtime errors
        + Means host and guest VMs must be running same JRE version etc.
        + I ran into bundles that couldn’t be started, e.g.
      * Providing bytecode classes directly as if already loaded a bit of a cheat, different from actually reading, possibly transforming and defining the bytecode
    - Better solution: configure holograph with mapping between guest file system directories and host directories, for read-only access, and execute actual ClassLoader logic that reads the bytecode from the zips/files
      * Requires implementing native methods around ZipFile, probably File too
      * To locate bytecode missing from heap dump, run ClassLoader.getResource(“<\*\*\*>.class”)
        + This requires HeapDumpClassMirror#getBytecode() to actually execute code, so it needs a reference to a ClassHolograph, which sucks because it breaks the clean stratification of mirror layers

Trying to set up circular reference to “the one” used at the API level, but having trouble avoiding infinite loops or NPEs…

* + - * + Trying different approach of leaving getBytecode() and several other methods unimplemented at the heap dump level and handling it all at holograph level

MUCH cleaner, but may have too much trouble with missing methods in heap dump graph (can’t do type checking for example…)

* Arrays
  + Each unique type generates a stub Java class that inherits ObjectArrayMirror
    - No instance methods no not necessary for the same reason as user classes, but needed for method overloading
  + Big issue: AALOAD/STORE opcode doesn’t specify expected element type the way method invocations do
    - Need to analyze bytecode to infer it!
    - Need bytecode version 1.6 or later - frame information required
      * No good, JRE only 1.5 even in later versions
    - Do analysis ourselves
    - Problem: type verification will trigger class loading, which needs a current thread, which may not exist in control flow of getMirageClass()
      * Solved by delaying first call to getMirageClass() until execution of MirageMethod.invoke()
* Special cases
  + All classes must extend Object, which declares toString()
    - Therefore all toString() methods must return actual java.lang.String instances
  + Object identity
    - How to define ==?
      * Could define ObjectMirror.sameObject(Object other)
      * Could allow mirages to be different, even if calling ObjectMirage.make() on the same mirror object twice
    - How to define hashCode()?
      * Supposed to be consistent across a single VM execution
    - How to define equals()?
    - Current approach: mirror identity implies mirage identity, lots of lazy maps to ensure same instances
  + Native methods
    - Enumerate replacements
      * Use AspectJ binaries even at this point (around execution)?
    - Automatically link all extra methods on ClassMirror#getNativeStubsClass according to idiom
      * How to deal with overloading? Stubs will all take ObjectMirrors instead of specific classes
        + Use annotations if this comes up
    - Any missing methods are stubbed with a method that throws an exception
    - Might need to easily support native methods outside JVM/JRE
  + Reflection
    - I intercept methods like Class.getFields()/newInstance()/getName() etc. to interface with mirrors instead
  + Enums
    - Just remove the flag – no need?
  + Soft/weak/etc reference
    - ???
* Optimization ideas
  + Cache consecutive field reads on “this” in same method?
    - No, not valid in light of concurrent writes
    - Maybe in specific well-understood/constrained cases
    - Or with clearer understanding of Java memory model
* Memory model
  + Mutable mirrors must handle concurrent access correctly
    - FieldMapMirror needs to be backed by ConcurrentMap
    - DirectArrayMirror needs to be backed by (array of AtomicReferences?)
    - Native methods like Unsafe.compareAndSwapInt() should call similar methods on FieldMirror
      * (in concert with Unsafe.objectFieldOffset)
    - Probably want “threadsafe” flag to save synchronization costs in common case

RETROACTIVE WEAVING

* Tried to reuse AspectJ pointcut parser, but not really suitable
  + Result (PointcutExpression) is opaque object designed only to match (fuzzily) joinpoint shadows, not useful model for other applications
* ABC parser works well, just added another entry point for parsing “pointcut\_expr” rule root
* Can re-use ajc aop.xml parser nicely
* Need to add AnnotationMirror model to mirrors model, so that weaving can read them from the VM’s classes and process advice
  + Nope – better idea: just load the raw annotations bytecode via Class#getRawAnnotations(), let Class.getAnnotations(s) do the hard work of instantiating them!
    - Missing native method anyway
    - Unfortunately also need to support constant pool functionality, which ASM generally encapsulates itself
    - Worked around using constant pool support exposed for custom attributes + a wee bit o hacking
* Configuration
  + Load-time weaving usually has three pieces
    - Where to find original classes
      * LTW can weave all classes (using agent), all standard class path classes (using aj or equivalent) or specific class loader classes (using WeavingURLClassLoader, or Equinox support)
      * With post-weaving no need to change as classes are defined, so all classes in the VM could be “woven” if desired
      * Best if possible to be consistent with any of the above scenarios
    - Where to find aspects
      * LTW usually has aspects in same class loader (WeavingURLClassLoader takes class path and aspect path as parameters, loads through both)
      * Equinox supports aspects in separate bundles
      * For post-weaving, could be already defined in image, or may need to inject using holograms
        + Again, could be in any classloader – defining if needed is orthogonal to finding them for weaving
    - Where to find configuration (aop.xml)
      * Normally one for each OSGi bundle (Equinox support)
      * Not yet sure if you would normally put one for each WeavingURLClassLoader
      * For post-weaving, need one for each operation of postWeave({classes}, {aspects})