Template Class Challenges

John Rose, JVM Architect

Burlington, March 2019

ORACLE

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.
It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Theory of JVM “template classes” and “species”

(as of the present moment)

- In JVM, template-ness is “really” constant-polymorphism.
  - I.e., a template class has holes in its constant pool.
- Holes are type-variables for parametric polymorphism.
  - Maybe holes could be numbers, strings, functions, etc.? (Cf. C++)
- Requires deep thinking about “what’s a constant pool”.
- Hardest problem: Avoid premature “code splitting”
  - Execute cold code from one set of bytecodes shared by species
  - The JIT inlines and customizes hot code, in the usual way.
  - Result: No footprint cost for seldom-used template instances.

Terminology
(built from from existing JVMS terms)

- “value object”, “reference object” (also value or reference instance)
- “value class”, “reference class” (also value or reference type)
- “interface class” (Object = honorary interface)
- object, class, reference: ***non-specific*** (non-primitives; ref. can be null)
- value: ***non-specific*** (includes all object references, null, all primitives)
- name (class, member), descriptor (field or method)
- resolution: a stable mapping from name to metadata (or error)
- “class template”, “method template” (even “field template”)
- “specialized class” (= "species" for short), “specialized method”
- generic parameter, hole (in template), variance (depends on hole)

Lots and lots of requirements

(some negotiable, most firm)

- Must mesh well with the existing frameworks of JLS, JDK, JVM[TI]
- Can improve on legacy code, replace old generics (migration)
- Initial sharing; unsharing in specializations happens late (linking, JIT)
- Arguments can be primitives, value types, specializations
- Can specialize value classes, reference classes, interfaces
- Can express variant/flat data, work with variant/flat arrays
- Specializations can be mentioned in field and method descriptors
- Can support wildcards (mild multiple/dual inheritance)
- Can have ad hoc optional members List<T>::compareTo(List<T>)
- Algorithms sort<String::compareTo>(…), size variance Vector<int,4>

What we (might well) understand now…

(basics)

- We should tweak the classfile **format**, rather than spin many classfiles.
- In L-world, every type variable has an Object **bound**.
- Object-bounded types work great with a-class instructions.
- **L-world** is our friend.
- **Primitives** are a minor corner case, best mediated by “P-box” VTs.

- “Codes like a class” is a good principle for templates as well as VTs.

What we (might well) understand now…

(type safety)

- Constant pool constants can be **resolved** to **metadata**.
- Resolution is **stable**, so a resolved constant is truly constant.
- Some constants are only **potentially** resolvable (field, method descrs.)

- The JVM **must** ensure type safety between callers and callees.
- …and similarly between overriding and overridden methods.
- Type safety is pairwise field-type equality.
- Field types are equal if they are resolvable to the identical metadata
- …“are resolvable” means either “did resolve” or “will resolve” (CLCs)
- CLCs ensure that, ***if*** a name resolves, it ***will*** resolve equal in 2 places

What we (might well) understand now…
(constant variance)

- The **constant pool** is where specialization happens (not bytecode).
- Some constants are **variant** and others are invariant.
- Constants inherently **depend** on other constants (except leaves).
- Variant constants are those which depend somehow on **holes**.

Note: language folks dislike this oddball use of the term *variance.*

What we (might well) understand now…
(CP segments)

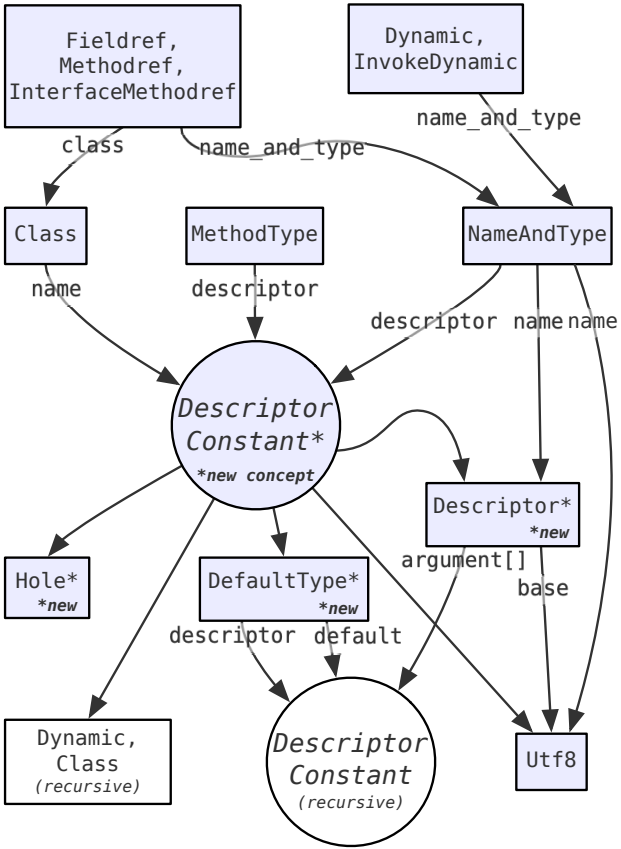
- **Same-variant** field, method, inner class, nested inside variant class.
- There are **layers** of variance, inducing a **segment tree** in the CP.
- **Hypervariant** generic method nest in variant class
- Also differently-variant inner class inside outer class.
- Extra variance segments for **optionality**.
- (Plus, remove variance from parts of those combinations.)

Note: We might start talking about a *scope tree* instead of a segment tree.

- CP segmentation and dependencies embody variance groups.
- CP variances can depend on condy.
- CP variances can depend on non-type values (behaviors, parameters)

On the bleeding edge

http://cr.openjdk.java.net/~jrose/values/template-jvms-4.4.pdf



On the bleeding edge

- CONSTANT_Descriptor(<*descr-string-with-#holes*>, <*t1*>, <*t2*>, …)
- species descrs. with hole: List<#>, Map<String,#>, Map<#,#>
- array descrs. with holes: “[#”, “[[#“, “[List<[#>” => Desc(“[#”, “List<[#”)
- method descr, with holes (##)#, (I#[J)Object

- We probably want a rule which replaces “stringiness” by “pointeriness”.
- All **resolved** parts of a descriptor should be given their own CP slots!
- Example (where “#X” marks a slot for X): (#List<Object>,#Foo,String)int
  …meaning, method taking a List of Files, Foo, and String.
  The List and the Foo can both be checked as resolved metadata.
  The stringy bits “Object”, “String” can rely on CLCs for type safety.

What we don’t yet understand fully…

type and member resolution

- Exactly how does CONSTANT_Class resolution proceed?
- Exactly how does CONSTANT_Fieldref resolution proceed?
- Exactly how does CONSTANT_Methodref resolution proceed?
- How do type variable (hole) bindings influence member resolution?

- How to use an unspecialized template as a plain (erased?) type?
- How to resolve the invariant members of an unspecialized template?
- Similarities and differences of class, interface, and template supers.

- How much bridging is needed?

What we don’t yet understand fully…

type checking

- Must verify type correctness inside one class file (verifier type system).
- Must preserve type correctness across separate compilation (linkage).
- Are flat string descriptors enough? Can we lean on them?
- How do we ensure that two occurrences of List<Foo> are the same?
- What happens when some Foo cannot be locally denoted by List?
- What happens when there are two equally valid proposed Foo types?
- How much extra system dictionary do we need? (scaly dragons)
- How many extra class loader constraints are needed? (Which loader?)
- Is there an alternative type check (method-handle-like) to CLCs?
- How does the verifier change if we adopt an alternative type check?

What we don't yet understand fully…

class file format

- Class file format can be done many ways (Bikeshed Alert!)
- Single CP vs. multiple distinct CP "islands"
- Indexing of constants: global numbering vs. "overlays"
- How to format variadic CP structures? (descriptors like condy?)
- How to organize linkage between segments and metadata items?
- (Bi- or uni-directional pointers? Physical co-location?)
- Nesting: Member-in-class or class-in-class + isolated method
- (Do you physically nest in the class file where you nested in source?)
- Prototype: One CP w/embedded segments + old-school metadata. (segment_info records after CP; class/field/method point to segments)

What we don’t yet understand fully…
(miscellaneous)

- Rules for translation strategies: When to erase?
- How to do ad hoc specializations? List<boolean>, Map<int,T>
- Selection (dispatch) a method which is both **virtual** and **hypervariant**.
- How to get primitives into the game?

- How to bootstrap methods play in this?
- Can a segment control its own setup (larval specialization state)?
- Are template parts reflected up to a BSM? (Control inversion?)
- Does this include reflection over bytecodes? (Lambda cracking?)
- How does the BSM tell the JVM how to assemble them?

Questions?

…Answers??