Adventures on the Road to Valhalla

(A play in at least three acts)

Brian Goetz, Java Language Architect

Prologue

Croaking Chorus of the Polywogs

(apologies to W. S. Gilbert, and to Aristophanes)

Why do we need better generics?

- Generics currently don’t deal well with primitives
- Users have always wanted **`ArrayList<int>`**
  - And have it backed by a real **`int[]`**
  - But instead, we have to use boxing (**`ArrayList<Integer>`**)
    - More footprint, worse locality
  - If we had to do that for value types, it would mostly defeat the purpose
- So, generics need to play nicely with value types
  - And primitives can come along for the ride

What’s the problem with generics?

- Generics in Java rely on *erasure*
  - Type variables are erased to their bound (usually Object)
- Generics over primitives *and* references run into several roadblocks
  - Supertypes: bound must be a supertype of all possible instantiations
    - No common supertype between primitive and reference types
  - Bytecodes: generic values are moved by \`a\` bytecodes (aload, astore)
    - There is no bytecode that can move both a ref and an int
- Expedient choice circa 2004: no primitive instantiations ☹
  - Today’s problems come from yesterday’s solutions…

Many paths to parametric polymorphism

- Parametric polymorphism is a tradeoff of *type specificity* vs *footprint*
- C++ uses compile-time template expansion
  - Great type specificity, lousy code sharing
- C# pushes type variables into the bytecode (parametric bytecodes)
  - Good type specificity and sharing, high VM complexity
- Java erases type variables to their bound
  - Great sharing, but doesn’t play well with primitives (and values)
  - Want to fix that

The Prime Directive

- Compatibility, compatibility, compatibility
  - Existing bytecode must continue to mean the same thing
  - Existing Java source code must continue to mean the same thing
  - Must be able to compatibly and gradually migrate “old” generic classes (and their clients, and their subclasses) to “new”
- At the same time …
  - Don’t impose Java language semantics excessively on the JVM

Act 1

Lives of Quiet Contemplate-tion

(apologies to H. D. Thoreau)

Generic class specialization

Our first attempt

- Compiler continues to generate erased classfiles
- Classfiles augmented with additional generic information
  - Ignored by VM, but can be used to produce specialized classes
- Used name-mangling technique to describe specializations
  - Temporary hack for prototyping – not a long-term plan
  - The name **`Foo${0=I}`** means “Foo with type var #0 instantiated with int”
- Class loader recognizes mangled names
  - Does specialization on the fly as needed

Specialization example

- A simple Box<T> class

```
class Box<any T> {
    T val;

    public Box(T val) { this.val = val; }
    public T get() { return val; }
}
```

- Erases to Box
  - T’s replaced with Object

```
class Box {
    Object val;

    public Box(Object val) { this.val = val; }
    public Object get() { return val; }
}
```

- Specializes to Box${0=I}
  - (Some) Object replaced with int

```
class Box${0=I} {
    int val;

    public Box(int val) { this.val = val; }
    public int get() { return val; }
}
```

Specialization metadata

- Specializing involves specializing signatures *and* bytecode
  - Must know which Objects to replace with int
  - Must know which aload to replace with iload
- Generic signature information is already (mostly) present in classfile
- Need to annotate bytecodes with type metadata
  - BytecodeMapping attribute
    - Maps bytecode at given index to specialization metadata for that bytecode
    - Brittle, but good enough for prototyping

Specialization metadata

Signatures (methods, classes, fields)

```
class Foo<any T> extends Bar<T> { ... }
```

```
class Foo extends Bar
Signature: #12 // <T:Ljava/lang/Object;>LBar<TT;>;
```

```
class Foo1${0=I} extends Bar${0=I} { ... }
```

Specialization metadata

Type 1 – data-movement bytecodes (aload, astore, …)

```
class Foo<any T> {
    T ident(T val) { return val; }
}
```

```
class Foo {
    T ident(T);
        0: aload_1
        1: areturn
    BytecodeMapping:
      Code_idx  Signature
          0:    TT;
          1:    TT;
    Signature: #18 // (TT;)TT;
}
```

```
class Foo${0=I} {
    int ident(int);
        0: iload_1
        1: ireturn
}
```

Specialization metadata

Type 2 – class bytecodes (new, checkcast, …)

```
class Foo<any T> {
    Foo<T> make() { return new Foo<T>(); }
}
```

```
class Foo {
    Foo<T> make();
        0: new #2 // class Foo
        ...
    BytecodeMapping:
      Code_idx   Signature
          0:     LFoo<TT;>;
}
```

```
class Foo${0=I} {
    Foo${0=I} make();
        0: new #2 // class Foo${0=I}
        ...
}
```

Specialization metadata

Type 3 – invocation and field access bytecodes

```
class Foo<any T> {
    T t;
    T get() { return t; }
}
```

```
class Foo {
    T get();
    0: aload_0
    1: getfield #2 // Field Foo.t:LObject;
    4: areturn
    BytecodeMapping:
      Code_idx  Signature
          1:    LFoo<TT;>;::TT;
          4:    TT;
}
```

```
class Foo${0=I} {
    int get();
    0: aload_0
    1: getfield #21 // Field Foo${0=I}.t:I
    4: ireturn
}
```

Specialization metadata

Type 4 – invokedynamic

```
class Foo<any T> {
    Consumer<T> m() { return t -> { }; }
}
```

```
class Foo {
    Consumer<T> m();
    0: invokedynamic #2, 0
    5: areturn
    BytecodeMapping:
      Code_idx  Signature
          0:    ()LConsumer<TT;>;::{0=(TT;)V&1=LFoo<TT;>;::(TT;)V&2=(TT;)V}
BootstrapMethods:
0: #35 invokestatic ...
    Method arguments:
      #36 (Ljava/lang/Object;)V
      #37 invokestatic Foo.lambda$m$0:(Ljava/lang/Object;)V
      #36 (Ljava/lang/Object;)V

}
```

Generic methods

- Generic methods can be invoked with indy
    - Bootstrap protocol can encode generic type arguments
    - Bootstrap method can do on-the-fly specialization
    - Specialized method wrapped in a container class
        - Loaded with defineAnonymousClass, host class = implementing class
- Static generic methods can be linked with a ConstantCallSite
- Instance methods must do dispatch computation to find target
    - Link to cached callsite
- Still, lots of fiddly complexity
    - Super calls
    - Desugared lambda methods

Other bits of "fun"

- Some bytecodes, like if_acmpeq, are messier to specialize
  - Bytecode set is not orthogonal – no if_icmpeq
- Renumber LVT slots when specializing with long/double
  - And hope to not run out…
- Accessibility bridges

```
class X<any T> {
        private T t;
        void foo(X<int> x) { … x.t … }
}
```

  - Here, accessing private field across class boundaries – but has to work!

Summary – Act 1

- On the fly template-based specialization works!
  - And is *compatible with the VM we have*
- So, a successful experiment?
- Well …
  - No nontrivial common supertype between Foo<int> and Foo<String>
    - Which means: no way to say “any instantiation of Foo”
    - Pain for library implementors
  - Terrible sharing characteristics
- Nothing here is impossible, but lots of small complexities
  - Death by 1000 cuts

Act 2

The Call of the Wildcard

(apologies to Jack London)

What about Foo<?>

- As much as people hate wildcards…
  - They apparently hate having their wildcards taken away even more!
  - Wildcards are often needed by implementations
  - Also used in APIs as an alternative to generic methods
- Wildcards heal the rift caused by heterogeneous translation
  - Just because Foo<int> and Foo<String> are represented by different classes (an implementation detail), they still have a common Foo-ness

What about Foo<?>

- If we have

```
class Foo<any T> extends Bar<T> { }
```

- Then we want

```
Foo<int>  <:  Foo<?>
Foo<int>  <:  Bar<int>
```

- So Foo<?> cannot be a class type (Foo<int> can’t extend two classes)
  - But Foo<?> is a class type today
- We’re overconstrained
  - Compatibility dictates that Foo<?> means Foo<? extends Object>
  - Intuition suggests that Foo<?> means “any instantiation of Foo”

Rescuing wildcards

- We've divided type variables into two categories – "erased" (legacy) and "any" (new)
  - Let's do the same with wildcards
    - Foo<any> -- Foo with any instantiation
    - Foo<erased> -- corresponds to current meaning of Foo<?>
  - And possibly deprecate the syntax Foo<?> (as it is now confusing)

Representing wildcards

How to represent wildcards in the bytecode?

- Continue to represent Foo<erased> as we do now – as erased type
- Prototype strategy: introduce a synthetic *interface* (Foo$any) to represent Foo<any>
  - Lift methods of Foo to Foo<any>, with boxing if needed
  - Lift accessors for fields of Foo to Foo<any>, with boxing
  - Lift supertypes of Foo to Foo<any>
- Make Foo<any> a supertype of all instantiations of Foo
  - Primitive/value instantiations may need boxing bridges

Translation with wildcards

```
class Box<any T> {
    T val;
}
```

```
interface Box$any {
    synthetic Object get$val();
    synthetic void set$val(Object val);
}
```

```
class Box implements Box$any {
    Object val;
    // obvious accessor implementation
}
```

```
class Box${0=I} implements Box$any {
    int val;
    // boxing accessor implementations
}
```

- Member access with concrete receiver (Box<int>, Box<String>) is translated directly, as today
- Access against wildcard receiver (Box<any>) is redirected through interface
    - Field access through wildcard redirected through accessor methods
    - Performance cost borne entirely by users of wildcards

Translation with wildcards

Boxing bridges

- Specializations will need boxing bridges to conform to the wildcard interface

```
class Box<any T> {
    T get() { ... };
}
```

```
interface Box$any {
    Object get();
}
```

```
class Box implements Box$any {
    Object get() { ... }
}
```

```
class Box${0=I} implements Box$any {
    int get() { ... }
    bridge Object get() { ... bridge to get()I ... }
}
```

More translation examples

- Translation of ref instantiations (including erased wildcards) is unchanged
- Translation of new types – primitive instantiation and any-wildcards – is new

```
class Box<any T> {
    Box<String> a;
    Box<int> b;
    Box<any> c;
    Box<?> d;
}
```

```
class Box implements Box$any {
    Box a;
    Box${0=I} b;
    Box$any c;
    Box d;
}
```

Wildcard challenge – accessibility

- What about protected and package-access members?
  - Classes can have them, interfaces can't
  - Need help from the VM here!
    - Private, package members in interfaces?
- We already have a problem with accessing private members across nests of inner classes
  - Specialization makes this worse; Foo<int> may want to access private members of Foo<Object>
  - Since there's only one source class Foo, this seems reasonable
  - Need help from the VM here!
    - Privileged cross-class access for nest-mates

Wildcard challenge – arrays

- What happens when a T[] shows up in a signature?
  - Can’t translate as Object[] … because an int[] is not an Object[]
- Need some help from the VM here…
  - Inject Array<int> as supertype of int[]
  - Inject raw Array as supertype of Object[]
- Array<any> is a supertype of both…
  - So Array<any> is a common supertype of Object[] and int[]
  - Translate uses of T[] as Array<any> in Foo<any>

```
interface Array<any T> {
    int size();
    T get(int index);
    void set(int index, T value);
}
```

Wildcard challenge – arrays

- Just as we use Object as the common supertype in the wildcard interface for T
  - We use Array<any> as the common supertype for T[]

```
class ArrayUser<any T> {
    T[] m() { ... };
}
```

```
interface ArrayUser$any {
    Array$any m();
}
```

```
class ArrayUser implements ArrayUser$any {
    Object[] make();
    bridge Array$any m() { ... }
}
```

```
class ArrayUser${0=I} implements ArrayUser$any {
    int[] make();
    bridge Array$any m() { ... }
}
```

Summary – Act 2

- Pros
  - More reasonable programming model
  - Excellent compatibility with existing code
- Cons
  - Still no code sharing between Foo<int> and Foo<String>
  - Needs more help from the VM to make this viable
- The language story here is actually pretty simple
  - Some type variables are decorated with “any”
  - Need to use “any” wildcards with “any” type variables
  - In the absence of “any”, *nothing changes*
  - Some operations (e.g., assignment to null) not permitted on any-tvars

Act 3

Sweet Sharity

(apologies to Neil Simon)

Sharing

- A key problem with the approach outlined so far is *code sharing*
  - If every specialization is a unique entity, this leads to lots of duplication
- Erasure gives us good sharing across reference types
  - One implementation represents many instantiations
- We’d like something similar for values
  - Maybe one set of native code per size (ArrayList<32bit>, ArrayList<64bit>)
- Push some knowledge of parametric polymorphism into the VM
  - But first, need to simplify our specialization transform
  - Act 1 transform is *way* too complicated!

Sharing

- To get more sharing, the VM needs to understand better how List<int> is related to List
  - If we have to modify every field declaration, method declaration, and bytecode in the implementation, this relationship is too complicated
- Strategy: consolidate all type information in the constant pool
  - Much of the type information is already there (e.g., method sigs)
  - There should be *one* place where the binding T=int is recorded
  - Turn specialization of *classes* into specialization of the *constant pool*
- Consequence: some types (e.g., parameterized types) are *structural,* not *nominal*
  - Need to undo some nominality assumptions in classfile format

Don’t erase so early

- Need to retain more generic type information in the constant pool
  - But don’t want to ask the VM to reason (much) about erasure
- New classfile forms
  - GenericClass attribute – registry of a classes type variables
  - ParameterizedType constant – a parameterization of a generic class
    - Plus a type signature to represent “erased”
    - Represent List<String> explicitly as List<erased>
  - TypeVar constant – represents a use of a type variable
  - MethodDescriptor – structural description of a method descriptor
    - Instead of the current nominal trick

Don’t erase so early

- New constants for type variable use and for type parameterization
  - Distinct constants for each type variable
    - Otherwise, can’t tell which uses of Object correspond to T, U, or Object
  - At every type variable use, statically precompute erasure

```
CONSTANT_TypeVar_info {
    u1 tag;
    u1 tvar;      // Index into class tvar table
    u2 erased;    // Type to be used if erased
}
```

Don’t erase so early

- Need a way to refer to a specialized class in bytecode
  - Mangled names are only good enough for a prototype
  - Specialized types are fundamentally *structural*
    - Bummer, all other classfile type descriptions are nominal
  - Represent Map<int, String> as ParamType[Map, int, erased]
- A Class constant can refer to one of these as well as a UTF8

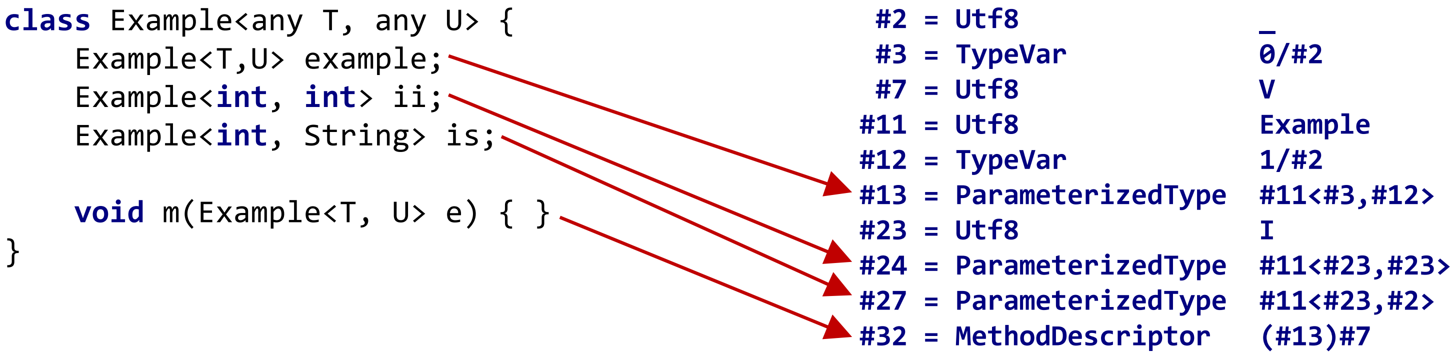
```
CONSTANT_ParameterizedType_info {
    u1 tag;
    u2 clazz;            // class being parameterized
    u1 count;            // how many tvars?
    u2 params[count];  // tvar instantiations
}
```

Don’t erase so early

Specialization procedure

- When we go to resolve a parameterization like Map<int, String>
  - This is described by a ParameterizedType
  - Create a specialization context containing bindings of tvars
  - Resolve TypeVar constants to ordinary UTF8 descriptors
    - With data from specialization bindings
  - Then resolve ParamType, MethodDescriptor, and ArrayType constants into ordinary nominal UTF8 descriptors
    - Via string interpolation
- And we have a specialized classfile!

Don’t erase so early



```
class Example<any T, any U> {
    Example<T,U> example;
    Example<int, int> ii;
    Example<int, String> is;

    void m(Example<T, U> e) { }
}
```

```
 #2 = Utf8                 _
 #3 = TypeVar              0/#2
 #7 = Utf8                 V
#11 = Utf8                 Example
#12 = TypeVar              1/#2
#13 = ParameterizedType    #11<#3,#12>
#23 = Utf8                 I
#24 = ParameterizedType    #11<#23,#23>
#27 = ParameterizedType    #11<#23,#2>
#32 = MethodDescriptor     (#13)#7
```

T=erased, U=erased

```
 #2 = Utf8       _
 #3 = Utf8       Object
 #7 = Utf8       V
#11 = Utf8       Example
#12 = Utf8       Object
#13 = Utf8       Example
#23 = Utf8       I
#24 = Utf8       Example${II}
#27 = Utf8       Example${I_}
#32 = Utf8       (LExample;)V
```

T=int, U=erased

```
 #2 = Utf8       _
 #3 = Utf8       I
 #7 = Utf8       V
#11 = Utf8       Example
#12 = Utf8       Object
#13 = Utf8       Example${I_}
#23 = Utf8       I
#24 = Utf8       Example${II}
#27 = Utf8       Example${I_}
#32 = Utf8       (Lexample${I_};)V
```

T=int, U=int

```
 #2 = Utf8       _
 #3 = Utf8       I
 #7 = Utf8       V
#11 = Utf8       Example
#12 = Utf8       I
#13 = Utf8       Example${II}
#23 = Utf8       I
#24 = Utf8       Example${II}
#27 = Utf8       Example${I_}
#32 = Utf8       (Lexample${II};)V
```

What about the bytecodes?

- Inconvenient fact: bytecodes are strongly typed
  - Need to change ‘aload’ to ‘iload’ when specializing for T=int
- If we want for specialization to operate only on the constant pool…
  - Then parametric bytecodes need to refer to the CP to get their types
  - How about a set of “universal” (parametric) bytecodes?

```
#3 = TypeVar[0]  // instantiation of tvar T
ureturn #3       // can be quickened by VM
```

- Refers into same CP slot as signatures that involve T
  - Verification can be performed against the template, rather than each specialization

What about the bytecodes?

- Inconvenient fact: bytecodes set is not orthogonal
  - For example, if_acmpxx bytecodes has no analogue for d or f
  - Have to use dcmp + if instead
- Need a few more bytecodes in our universal set (e.g., ucmp_eq)
  - Use this for specialized comparisons
- At this point, type-1 bytecodes can be specialized just by operating on the constant pool!

What’s the point?

- All of these representational changes are in aid of enabling the VM to treat a specialization like List<int> as a projection of List
  - While sharing most of the representation between all projections
  - Without having to understand the language-level generics system
- So, how does the VM resolve a Class constant whose payload is a ParameterizedType?
  - Dumb implementation could just run the same specialization as current prototype
  - But needs some help from the language runtime
  - And some help from reflection

Summary – Act 3

- Pros
  - Reasonable programming model
  - Excellent compatibility with existing code
  - Path to high degree of sharing
    - Also admits “dumb” V1 implementation without much sharing
- Cons
  - Java generics will be a half-erased / half-reified hybrid
    - References erased, values reified
      - The price we pay for compatibility!
    - But VM won’t have to understand this
    - Other languages can use specialization to fully reify

Curtain