Fortress function/method/type encodings

David Chase

February 6, 2011

This note describes the manglings applied to various Fortress entities in their translation to the JVM. An outline of the classloader behavior triggered by these encodings (if any) is also included. Some of these manglings include what can only be called silly artifacts of the Fortress compiler's implementation history; some of these may be fixed over time.

1 Special characters

Table 1: Special characters used in generated code.

left oxford	⟦	begin generic parameters (can nest)
right oxford	⟧	end generic parameters (can nest)
danger snowman	⚠☃	void; used in Arrow types
heavy x	✖	schema marker; blocks subsequent rewriting
gear	✿	generic function marker
up index	☝	self type within generic methods
right index	☞	index indicator for functional method forwarders
envelope	✉	closure indicator

2 Dangerous characters and basic name encoding

Important note: at this stage of the Fortress implementation, there is no linker, and thus with the exception of Executable, components are expected to have the same name as the api that they export. This will change.

A Fortress component or api name containing at least one dot is encoded as a package and class; a name with no dots is encoded as a class with no package. For example, "`a.b.c`" encodes as package "`a.b`", class "`c`". Within the JVM, this name would appear as "`a/b/c`".

Non-generic top-level functions of an api or component appear as static functions of the class for the api/component in which they appear. Their names may be further mangled.

Objects of an api or component appear as if they were inner (final) classes of the class for the api/component in which they appear. Their names may be further mangled in certain cases. For example, object O within "`a.b.c`" has the JVM name "`a/b/c$O`". Their constructors appear as static methods of the component, as if declared with the same name (in the absence of other manglings). Singleton objects appear as static final fields of the api/component class, with no constructor (or rather, the constructor name is mangled and only referenced in the static init of the class). This story is somewhat complicated by generics.

Traits of an api or component appear as an inner interface and inner abstract class (two names) of the class for the api/component in which they appear. The interface describes the methods of the trait, and the abstract class extends the interface, provides static methods implementing any default methods of the trait, and also provides forwarding (instance) methods to those defaults for convenient inheritance whenever the trait appears in first position of an extends clause. The abstract class for a trait, extends the abstract class for the first trait that the trait itself extends. Like objects, the name of the interface can be the same as the Fortress name – for example, trait T within "`a.b.c`" has the JVM name "`a/b/c$T`". The abstract class is encoded as if it were an inner class of the trait's encoded interface, with name "DefaultTraitMethods". Extending the example, "`a/b/c$T$DefaultTraitMethods`". This encoding was chosen primarily because it eased bootstrapping; traits and objects for primitive types could be hand-coded in Java and then compiled to bytecodes.

Throughout this description of the name encoding, dangerous characters appear. All names and descriptors are subject to a special mangling that removes dangerous characters. The official dangerous characters transformation is described at John Rose's blog (`http://blogs.sun.com/jrose/entry/symbolic_freedom_in_the_vm`), but it is not applied blindly. At the top-level of a name (outside of encoded type parameter lists) dangerous-characters mangling is applied to $-separated segments; within generic parameter lists, it is applied to the entire list, including any $, /, or ; that may appear in the names there.

3 Generic traits and objects

A reference to a Fortress generic trait or object is encoded as the encoded trait name, followed by a left oxford bracket (⟦), then a semicolon-separated list of the encoded static type arguments, and then a right oxford bracket (⟧). For example, within Fortress component "hi" containing generics `C⟦T,U⟧` and `bo⟦T⟧` and trait A, a reference to `C⟦A,bo⟦A+⟧⟧`

is encoded as

`hi$C⟦hi$A;hi$bo⟦hi$A⟧⟧`

which after dangerous-characters mangling becomes

`hi$\=C⟦hi\%A\?hi\%bo⟦hi\%A⟧⟧`

At compile time, each generic trait is converted to a template class, with the name of the generic trait equal to the stem followed by left Oxford, semicolon-separated static parameter names, and right Oxford. The template it stored in a ".class" file with a different name, that keeps the Oxford brackets but omits the static parameter names. Accompanying that file is a second file with suffix ".xlation" containing the names of the static parameters. The parameters are stored as a list using an extremely simple serialization protocol. For "hi.C⟦T,U⟧" in the example above, the template file is named `hi$C⟦⟧.class` and contains (dangerous-mangled) class `hi$\=C⟦T\?U⟧`. The accompanying file `hi$C⟦⟧.xlation` contains (exactly) "`LIST_1.0 STRING_1.0 2 1 T 1 U`", indicating version 1.0 of the list and string serializations, a List of 2 items, a String of 1 character T, a String of 1 character U.

For generic traits, the DefaultTraitMethods is encoded as if it were an inner class of the generic, and thus also appears as if it were generic. For trait "C" in this example, the corresponding class is `hi$\=C⟦T\?U⟧$DefaultTraitMethods.class` stored in `hi$C⟦⟧$DefaultTraitMethods.class` and the static parameter information is stored in `hi$C⟦⟧$DefaultTraitMethods.xlation`

At run time, any reference to one of these generic-encoding classes triggers special behavior in the Fortress class loader. The static arguments in the reference are stripped out, to yield the basename of the template file and the .xlation file. Those files are read as resources, not classes. The names of the static parameters and values (text) of the static arguments are paired up in a map, and then the "bytecodes" in the template are processed with ASM to perform a context-sensitive replacement of parameters with arguments. The result is the class that is actually loaded. The loaded class may contain additional generic references, which are in turn expanded. *Interesting to consider, could we ever accidentally deadlock our classloader from two threads?*

When a greater variety of generic parameters are supported (nat, bool, opr, dimension, unit), the information in the .xlation files will be extended to keep track of the parameter category. This may lead to different context-sensitive rewriting, but given the infrastructure already in place, this should not be difficult.

4 Arrow types

An arrow type is both a string appearing within other identifiers, and a type in its own right. A Fortress function with domain *domain* and range *range* will have corresponding Java type Arrow⟦$\widehat{domain}$; $\widehat{range}$⟧ in the generated code, where $\widehat{domain}$ is the Java class/interface

(tuple) encoding for the Fortress domain (tuple), and $\widehat{range}$ is the class or interface for the Fortress result type. If the Fortress type is void ("()"), then the encoded type is ⚠☃; if it is a singleton, then it is just the encoding of that type; if the domain type is a tuple, then the encoded elements of the tuple type, separated by semicolons. Note that tupled result types are not unwrapped, and appear as a encoded Tuple type (not yet implemented, but trivially Tuple⟦$\widehat{element1}$; $\widehat{element2}$; ...⟧).

5 Closures

Closures, meaning, function values derived from the naming of a top-level function, are compiled into a reference to an oddly-named class, which triggers code generation when the class is loaded.

The general form of the class name is *packageClass*$*function*✉$Arrow⟦$\widehat{domain}$; $\widehat{range}$⟧ where

- *packageClass* is the Java package and class derived from the Api or Component containing the top level function.
- *function* is the name of the function itself.
- $\widehat{domain}$ is the translation of the function's domain
- $\widehat{range}$ is the translation of the function's range

For example, a non-call reference to

```
printlnZZ(n:ZZ32):()
```

defined in component C6 results in a load of static field "closure" from

```
C6$printlnZZ✉$Arrow⟦com/sun/fortress/compiler/runtimeValues/FZZ32;⚠☃⟧
```

Dangerous-characters-mangled, this is

```
C6$printlnZZ✉$\=Arrow⟦com\|sun\|fortress\|compiler\|runtimeValues\|FZZ32\?⚠☃⟧
```

Here, the package is empty (top level) because the Api name is undotted. The class name is "C6", and the function name is "println". "com/sun/fortress/compiler/runtimeValues/ZZ32" is the Java/JVM type for Fortress ZZ32, and the function returns void, so the range is "⚠☃".

The Fortress classloader checks for an occurrence of "✉" in a requested class, and if one is found, then a closure class is synthesized based on the information in the name itself.

6 Generic functions

Generic function references are also compiled into references to an oddly named class. The generic function class is also a closure. The form of the generated name is

$$packageClass✲\$function⟦static\_arguments⟧✉\$✖\text{Arrow}⟦domain;range⟧$$

Unlike closures, the class resulting from a reference is not synthesized from just its name, but is instead created by expanding a template "class" where the static parameters appear in the place of the static arguments (just the names, no extends annotations). The template is stored in a ".class" file with basename

$$packageClass✲\$function⟦⟧✉\$✖\text{Arrow}⟦domain;range⟧$$

Note the lack of static parameters in the signature; this is actually a lurking bug, once we implement overloading of generic functions. The expansion of a generic function proceeds exactly the same as the expansion of a generic trait or object.

The packageClass and function are the same as in closures; the static parameters are those supplied in the definition of the function. The schema serves to disambiguate different generic functions overloading the same name; currently it is just the arrow type of the generic function, as declared (but this is not sufficient, that is the lurking bug). The ✖ character serves a special purpose; it blocks further translation within a name occurring with a template that is expanded by the classloader.

The purpose of the oddly named container file and the non-rewritten schema is to help work around the lack of a linker in the current implementation, and to simplify linking when one is finally needed. Ideally (for the code generator and classloader, if not for error reporting and humans trying to read generated code) static parameter names would be normalized at both declaration and reference, but currently they are not. Lacking that, the Fortress classloader needs a way to find the template for a given generic function. It does this by stripping out the static parameters and using the name that results, together with the schema for the function. The schema is not rewritten, and thus matches between definition and use sites (assuming some consistency in static parameter names between the api's declaration and the components definition of the same generic function). Note that the schema has no particular semantic meaning; it merely serves to disambiguate multiple generic functions with the same name, and it could in principle be the source location in the api declaring the function (or component, if it is not exported to an api).

The Fortress classloader checks for an occurrence of "✲" to detect generic functions. This check occurs before the closure test.

For example, the generic function `g[\T extends ZZ32, S\](x:T):ZZ32` in component C11 generates .class and .proto files with the mangled basename

`C11✲$g⟦⟧✉$\=✖Arrow⟦T\?com\|sun\|fortress\|compiler\|runtimeValues\|FZZ32⟧`

which unmangles to

C11✲$g⟦⟧✉$✖Arrow⟦T;com/sun/fortress/compiler/runtimeValues/FZZ32⟧

Note that closure bits for the generic function are not synthesized at run time, but instead come from part of the generic template. (However, the same code stamps out templates and synthesizes closures, it is simply called from different contexts).

6.1 Modifications to incorporate overloading of generics

When an overloaded function includes dispatch to (and instantiation of) a generic function, that will require two things. First, there will need to be (1) a symbolic instantiator (similar to what is done for generic methods), (2) a dispatch cache for the particular function (like generic methods), and (3) the generic that is instantiated must include an entrypoint in which type parameters have been erased, followed by casts to the instantiated types. This is so because the symbolic reference, being symbolic, cannot include a cast to a non-constant (symbolic) type.

7 Functional methods

A functional method in a trait or object is promoted to a top-level trampoline function, and the method name is rewritten by appending ☞K, where K is the index of *self* in the parameter list of the functional method. For example, the method f in trait T below

```
trait T
  f(x:String, self, y:ZZ32):String
end
```

is rewritten to the new method `f☞2(x:String, y:ZZ32):String` and the top level function is almost-as-if

```
f(x:String, renamed_self:T, y:ZZ32):String = renamed_self.f☞2(x,y)
```

What prevents this from being the exact translation, is the possibility of Fortress ASIF being applied to arguments to the functional method; for dispatch purposes, this alters the apparent type of a value, but the alteration only lasts for a "single" dispatch. In the case of functional methods, the altered type is in force until the body of the final method (after any method overloading) is reached.

8 Renaming of overloaded functions

Within any component, there may be overloaded functions. In some cases, the overloaded function may have the same domain as one of the functions in the overload. When this occurs, the overloaded function gets the canonical name that the function will be known

by if it is exported, and the name of the function in the overload that clashes is mangled by appending $SINGLE to its name. It is correct, though not necessarily efficient, to refer to the overloaded function even when it is known that only the single function is applicable, because dispatch will still arrive at the proper place. Within the overloaded dispatch itself, however, the distinction matters.

Note that, because no top-level function in an overloading is allowed to be more specific than a functional method, this single-mangling will never apply to a trampoline function for a functional method. (The overloading can be resolved at the method level.) **This is not true for the new overloading story; this needs to be corrected.**

Note that there may be a problem with methods and ASIF, if we lazily generate dispatch code for ASIF references, because we cannot lazily add methods to a class.

9 Functional methods of generic traits and objects

10 Generic methods

Generic methods are compiled very differently from ordinary methods, and also have a different calling convention. Because the JVM does not allow new methods to be added to an already-loaded class, these cannot be template-expanded as methods. Instead, each generic method leads to the creation of a lookup method, a generic top-level function, and a static lookup table. The lookup method has the same name as the generic method, but always takes two parameters, the first a hashcode of a string, and the second parameter that string, a representation of the static arguments to the generic method instantiation.

The lookup data structure and sample code are as if they were compiled from this Java:

```
private static BAlongTree sampleLookupTable = new BAlongTree();
public Object sampleLookup(long hashcode, String signature) {
    Object o = sampleLookupTable.get(hashcode);
    if (o == null)
         o = InstantiatingClassloader.
             findGenericMethodClosure(
                  hashcode,
                  sampleLookupTable,
                  "template_class_name",
                  signature);
    return o;
}
```

The generic top level function has an additional first parameter for self; after it is instantiated, it is returned and called as if it were a top-level function. The static arguments passed to the lookup function and used to instantiate the top level function are modified to

include a type argument describing the static type of self at the call site. This is necessary because of limitations of the JVM and its type system. The Object returned from the lookup method must be cast to an Arrow type so it can be invoked. However, which Arrow type is it? Recall that function types (Arrows, in this context) are contravariant in their domain type, and this cannot easily be encoded directly into the JVM's type system (Integer extends Number extends Object, but Object→... extends Number→... extends Integer→...; it requires global knowledge to figure out the upside-down Arrow hierarchy, and most of the casts are probably never performed). Therefore, by convention, Fortress Arrow types are invariant in their domains *when regarded as Java types, which is what matters for JVM verification*, and an actual Fortress subtype test will be open-coded to check the domains for appropriate relationships. This means that the static type of self and the type of the self parameter in the returned closure must match. *Strictly speaking, could be assignable-to, but the upper bound of all the traits declaring the generic method might not be representable, therefore this route was not pursued.* By providing a type parameter to specify the exact self type for the Arrow, a match can be ensured. *Note that this means that casting of arrow types may result in generation of wrapper code to both make the JVM happy and to apply the appropriate casts to the arguments when they are passed in.*

The generated closure is slightly modified from the usual, because its "apply" method contains a downcast to the known actual type of the trait or object defining the generic method.

The calling convention for a generic method closure is as follows:

1. evaluate the object expression
2. DUP the object value
3. push the hashcode and string for the static-self-prepended generic arguments. Note that in a generic context, these generic arguments may mention arguments bound in an outer instantiation; in that case special methods are used that are replaced with the appropriate instructions when the outer generic is template-instantiated.
4. invoke *object.methodName*
5. cast the returned Java Object to the appropriate Arrow type (same as original method signature, but with static self prepended.
6. SWAP closure and object
7. evaluate arguments
8. invoke closure

There is a naming convention for the closure that provides the body of the generic method, but the name is not directly exposed to referencing code. The compiler and the

`findGenericMethodClosure` method of `InstantiatingClassloader` are the producer and consumer of the name. As an example, suppose that component `C16` contains a trait `T` with a generic method `f⟦S⟧` mapping `S` to `Foo`.

```
component C16
trait Foo end
trait T
  f[\S\](s:S):Foo
end
```

The closure name is
`C16✵$T✌f⟦☝;S⟧✉$✖Arrow⟦S;C16/Foo⟧`
or with dangerous-characters mangling
`C16✵$T✌f⟦☝\?S⟧✉$\=✖Arrow⟦S\?C16\|Foo⟧`

This would be a great place for an example invocation.

11 Special methods

Certain methods are rewritten to loads of long and String constants when the templates containing them are instantiated. This is necessary to correctly rewrite invocations of generic methods occurring in a template (generic) class.

These are encoded by use of a magic class name ("CONST") and whatever constant encodings we decide we need; in principle, we could embed a Lisp interpreter here. Currently, the encoded method names are `String.`*stringValue* and `hash.`*stringValue* (yes, the capitalization is inconsistent, sigh).

12 Generic methods of generic traits and objects

The calling convention for a generic method of a generic trait/object is the same as the convention for calling a generic method of a plain trait/object. It has to be, since a generic trait could extend or be extended by a non-generic trait supplying the same generic method. The (generic) trait provides a lookup method, and the lookup method provides an appropriate closure, which must be cast in the same way before invocation.

The only interesting difference is that the generated closure will have a concatenated set of generic parameters; first the static self type, then the method static parameters, finally the trait static parameters (this is different from the order used for functional method declarations in a double-generic context, the goal is to cut down on annoying parsing of generic parameter lists). The hash code and generic method parameters passed to the lookup method will be the same, so the extra static parameter concatenation occurs within an extended closure lookup method.

```
private static BAlongTree sampleLookupTable = new BAlongTree();
public Object sampleLookup(long hashcode, String signature) {
    Object o = sampleLookupTable.get(hashcode);
    if (o == null)
         o = InstantiatingClassloader.
             findGenericMethodClosure(
                  hashcode,
                  sampleLookupTable,
                  "template_class_name",
                  signature,
                  trait_signature);
    return o;
}
```

Note that the hashcode and index strings do not depend on the trait signature; it is used only in locating the correct closure template to instantiate.

13 Functional generic methods

The forwarding function created for a functional generic method is itself a generic function, following the same naming conventions for a top level generic function of that name. The dotted method invocation within the forwarding function, follows the recipe for invocation of a dotted generic method.

It appears that the code generator, as of revision 4668, is failing to add the finger-index annotation to generic functional methods.

14 Tuples (and casting)

In the normal case, fortress tuples used as "arguments to function calls" will not result in heap allocation, but it can be handy to return tuples, and they may appear in certain generic situations. One problem with tuples is that it is not practical to encode all possible tuple subtype relationships into Java classes or interfaces, because there will be so many. For example, if the N elements of an N-tuple each have K transitive supertypes (including themselves), then there are K^N transitive supertypes of the tuple.

This means that, though Fortress type checking will be happy with

```
a:ZZ32 = 1
x:(Number, Number, Number) = (a,a,a)
```

the naively compiled Java will not pass verification, because `Tuple\=⟦ZZ32\?ZZ32\?ZZ32⟧` will not extend `Tuple\=⟦Number\?Number\?Number⟧`.

To deal with this, Fortress Tuple types provide the following:

- (shared) An abstract root class `Tuple` providing a method for determining its size and a method for getting its elements by index. Ideally, this would be an instance of Java `List`; this works as long as we are not unboxing carelessly (Fortress integers are not, naively, "int"). This might extend Java `AbstractList` (it also has to be a Fortress `Any` type, Java interface `Fortress$AnyType$Any`). *This type seems to do nothing more interesting than ordinary* `AbstractList`*, so for now, it will be* `AbstractList`*.*

- (shared) A super interface `AnyTuple`N (mangling TBD) where N is the number of elements in the tuple type. `AnyTuple`N declares methods `o1` through `o`N returning `Fortress$AnyType$Any` for accessing the elements without knowing their types. Note that in the bytecodes, these will appear as instance methods, with signature "`()LFortress$AnyType$Any;`".

  `AnyTuple`N extends `java.util.List`.

- (shared) A super class `AnyConcreteTuple`N (mangling TBD) where N is the number of elements in the tuple type. `AnyConcreteTuple`N supplies methods `o1` through `o`N returning `Fortress$AnyType$Any` for accessing the elements without knowing their types. Note that in the bytecodes, these will appear as instance methods, with signature "`()LFortress$AnyType$Any;`". `AnyConcreteTuple`N extends `AbstractList` and implements `AnyTuple`N `AnyConcreteTuple`N provides implementations for `AbstractList`'s size() and get() methods.

- An interface `Tuple...` parameterized appropriately. This implements `AnyTuple`N, ande declares accessors for the elements, `e1` through `e`N. Note that these will be properly typed and will therefore have signatures that depend on the element types.

- An implementation class `ConcreteTuple...` parameterized appropriately. It extends the appropriate `AnyConcreteTuple`N as well as the `Tuple...` interface with matching parameters.

  - A static factory method `make` taking N properly-typed elements, invoking the constructor.
  - A static factory-as-needed method `cast` for converting another tuple whose elements subtype (respectively) the elements of this tuple type, into this tuple type. This is accomplished by first checking for subtype and returning the same if it exists, otherwise invoking the constructor on the cast elements. This can throw ClassCastException; it is assumed that the caller knows what they are doing.
  - A static boolean method `isA` determining if another tuple could successfully be cast to this one.
  - Accessors for the elements, `e1` through `e`N. Note that these will be properly typed and will therefore have signatures that depend on the element types.

– Implementation for accessors defined in `Tuple`N, `o1` through `o`N.
– A constructor taking the (properly typed) elements as parameters.

Tuples are cast by invoking the `cast` method of the cast-to type; either the cast in place, or the tuple is effectively coerced (which is ok, since it is a value type).

15 Casting Arrows

Arrow types have the same mapping-relations-to-Java problem that tuple types do, only worse, because of contravariance. A function with type `Any->()` is a subtype of all other function types returning (); this cannot be statically encoded in the JVM type system, since new types can appear as generics are instantianted, and the JVM requires a fixed set of supertypes for any type.

For a given cast-to arrow type, `T->U`, and a value f of type `S->V` (S is a supertype of T, U is a supertype of V), the cast is accomplished in the following way:

- interface `Arrow⟦S;V⟧` extends `Arrow⟦java/lang/Object;java/lang/Object⟧`; thus, any Arrow type can be treated as an untyped arrow.
- `AbstractArrow⟦S;V⟧` includes an apply method that takes Java Objects and returns a Java Object. Within that apply method, are the necessary casts. *Question: will tuples of objects be an issue?*
- The `WrappedArrow⟦T;U⟧` type provides a consructor that takes an `Arrow⟦Object;Object⟧` and returns a new `AbstractArrow⟦T;U⟧` instance. The parameter(s) into the wrapper (which supports apply, etc) are passed to the object-typed parameters; the result is cast to the return type `U` (this will be a real Java cast that always succeeds, and analysis might discover this).

16 Overloaded function dispatch including generic functions

Initially, the (revised) model for overloaded function dispatch is to topologically sort the member functions from more-to-less specific, and iteratively attempt to match the functions, choosing the first winner. Static analysis ensures that the last function in the list needs no testing; it must match. This approach is taken so that the details of "more specific than" are centralized in static analysis. The first attempts to optimize this will merely look for CSEs among the constraint checks, and perhaps modify the order of constraint checking in order to perform common tests first.

The generated code for an overloaded function must perform three subtasks. First, it must verify constraints. This is one place where mistakes can be made; the code generator needs to be sure that no constraint is overlooked. In this overloading, the first function is more specific.

```
f[\T, U extends ZZ32\](t:T, u:U):U
f[\T, U \](t:T, u:U):U
```

This can get more complex. Consider this overloading

```
f[\T, U extends ZZ32\](t1:T, t2:T, u1:U, u2:U):U
f[\T, U \](t1:T, t2:T, u1:U, u2:U):U
```

Here, T is the join (more general) of t1 and t2, and U is the join of u1 and u2. The first function applies whenever the second (U) join is more-specific-than-or-equal to `ZZ32`. Note, however, that determining applicability does not require computing the joins; if the types of u1 and u2 are both extensions of ZZ32, then their join is also an extension of ZZ32.

The second task (itself a subtask of the first, sometimes) is to infer static parameters. Different contexts will be either invariant, covariant, or contravariant; invariant is easiest (if a parameter x has declared type `Invar[\T\]`, then if x is some sort of `Invar`, then T must be exactly that `Invar`'s static parameter). Top level parameter types are covariant, arrow ranges are covariant, and arrow domains are contravariant.

The third task (for generic targets) is to symbolically instantiate and invoke the targeted generic. This is similar to the technique used for generic methods, but slightly different. Generic function implementations must be enhanced in two ways; they must provide an entrypoint supporting symbolic instantiation with type parameters and hashcode (similar to generic methods), and the closure returned must supply an apply method where the type parameters of the function have been erased (because the invoking can only cast as far as the erased type). That apply method will contain the necessary casts, forwarding to the "real" apply method.

To infer static parameters, it will be necessary to add additional Fortress-specific type reflection operations. For any generic trait or object `TO[\U1, U2, U3\]`, the implemented "Fortress type" (distinct from the Java class of its implementation) must supply methods to obtain the first, second, and third static parameter types. The Fortress type must also supply methods, indexed by erased type, of the Fortress types that it extends (these will be constants if the extended type is not generic). (Implementations of) Fortress objects must supply a method to return their Fortress type.

What methods should all Fortress types support?

getExtends returns a list/array of Fortress types that this type extends

isGeneric

getName

getStaticParameters returns a list/array of the static arguments for this type. (What's a "static argument"? Does this include both parameter name and argument binding? Note that these are not always types.)

For purposes of dispatch, we need additional methods, and we need a system of naming these types. Suppose that the function whose contraints we are verifying has this signature, where List and Set are invariant in T.

```
f[\T extends Number\](l:List[\Set[\T\]\], a:T, b:T):List[\T\]
```

Assuming that the arguments l, a, and b have actual types L, A, and B, the following conditions must all be satisfied:

- `L <: List[\?\]`
- `L.asList#1() = Set[\?\]`
- `T <: Number` where `T = L.asList#1().asSet#1()`
- `A <: T` where `T = L.asList#1().asSet#1()`
- `B <: T` where `T = L.asList#1().asSet#1()`

One problem here is that this sequence of tests does not follow obviously from the signature. If the first (l) parameter were missing, the constraint on T is that it be equal to the join of A and B; however, because the first parameter produces an invariant constraint, we know the type that T must be. Note that the actual constraints are that both A and B extend T, and that T be the most-specific type with this property.

To compile the queries, note that sometimes no value of a type is available for querying in the style of Java `instanceof` (this is the motivation for `List[\Set[\...\]\]` – it is possible to ask parameter l if it is an instance of List-something, but to what should the instance-of-Set-something query be directed?)

Another thing to note is that if we pursue this implementation, it bakes in the restriction that a generic stem can only appear once in the transitive-extends of a class. If `asList#1` returns a single type instead of a set of types, then `List` can only appear once in the transitive extends.

Use of instanceof for stem queries suggests that the type system will take the form of two parallel interface and interface-implementation type hierarchies.

For sake of example, suppose that `Array[\T\]` extends both `Map[\NN, T\]` and `List[\T\]`. In the generated Java, the run-time type information will include three interfaces

Array_I Extends FRTTI, Map_I and List_I. Declares methods:
`asArray#1`

Map_I Extends FRTTI. Declares methods:
`asMap#1`
`asMap#2`

List_I Extends FRTTI. Declares methods
`asList#1`

and three interface implementations

Array_C implements Array_I

```
constructor and fields Array_C(t1 FRRTI)
asArray#1() = t1
final Map_C map = new Map_C(ZZ32_C.only(), t1);
final List_C list = List_C(t1);
asMap#1() = map.asMap#1()
asMap#2() = map.asMap#2()
asList#1() = list.asList#1()
```

Map_C implements Map_I

```
constructor and fields Map\_C(t1 FRRTI, t2 FRTTI)
asMap#1=t1
asMap#2=t2
```

List_C implements List_I

```
constructor and fields List\_C(t1 FRRTI)
asList#1() = t1
```

It appears that the extends list for a type must be initialized lazily, because the type itself can appear there, and also that each generic type instantiation must be unique (because of cycles through the extends list). All of this may occur in a multi-threaded context, so (in order to obey Java memory model rules) some of these fields (in particular, the extends items) must be volatile so that double-checked locking will work for initialization, and no locking will work for reading.

16.1 Implementation notes

It might be simpler to dispense with the FRTTI base type; if it has no methods, what's the point?

It looks like the names of the interface and class will be derived from the name, followed by "$RttiI" and "$RttiC".

The Java class created for each object must include a static member containing the type and instance method returning that type.

What do we do about self types? We need to be sure that cycles are ok. Consider

```
ZZ32 extends BinOp[\ZZ32, PLUS\]
```

What this means is that the naively specified constructors simply will not work; the constructor needs to take a thunk that will evaluate to a type, instead. Or, the type needs to defer creation of its supertypes.

```
class ZZ32.RttiC implements ZZ32.RttiI {
  static public ZZ32.RttiC only = new ZZ32.RttiC();
  Map.RttiC _BinOp;
  Map.RttiC asBinOp() {
     if
  }
}
```

Note, also, that we think we DO allow extension of multiple instantiations of the same generic with different opr parameters, for example

```
ZZ32 extends { BinOp[\ZZ32, PLUS\], BinOp[\ZZ32, TIMES\] }
```

A corollary of this is that it should not be necessary to infer an opr parameter dynamically (because this would be ambiguous, at least using the current approach). But this means that we need a name mangling that accounts for this.

17 Odds and ends

One thing to be aware of is that covariant type relationships (not part of Fortress yet except for Arrow and Tuple types) may be costly if mapped directly onto Java types. Covariant generics with Any-bounded static parameters (for example, `AnyList[\ T extends Any \]` will create special headaches because Tuple and Arrow type relationions are far too large to encode into the Java type system. One possibility will simply be to forbid that case; otherwise, it can be implemented by wrapping values, and using a method to implement strict equality, not just a pointer comparison.