Nomenclature

Nomenclature is important to a JVM MOP since the main programmable channel in `invokedynamic` is a name string. Based on the mangling convention described earlier, we will used colon delimiters to structure names into simple token sequences, and use those structured names to manage linkage across the JVM and across languages.

Top-level name syntax

Here are some key questions in designing such names.

1. Should the names be reasonably pleasant to read (modulo mangling effects, which can be hidden in polite company)? Default answer: yes. The dynamic Java design sketch in this wiki uses simple keywords, and they seem to read well.

2. How does a language (or other MOP player) reserve a part of the namespace? Is it enough to say that certain early adopters get short-hand prefixes like `ruby:`, `python:`, etc? Probably. Eventually we'll need something like fully qualified names, like `:com.acme:acmeslang:`

3. Does the notation allow elision? For example, if every MOP-name in a compilation unit belongs to AcmeSlang, maybe we can declare that `acmeslang:` is optional. But, there should be (a) a way to escape to other name spaces, and (b) no need for an escape to access standard prefixes like `method:`. Another way of phrasing this is, can you say the equivalent of `import acmeslang:*`, issue names like `event:foo` to mean AcmeSlang events, and then issue fully qualified ZeugmaChat names like `:com.zeugma:event:bar`. I thinks this kind of importing or context sensitivity is important, that conciseness of bytecodes is a medium-important goal.

3. What are the common notions that can be permanently imported? Anything that the JVM defines natively needs a name. This includes `method:` and `field:` and `element:` and perhaps `class:` (if a class or interface can be an operation--and if `class:x` then also just `int`, `boolean`). Certainly `new:`. Certainly the distinction between a static, a special, and a normal method name. Maybe array element getters and setters Maybe also `set:field:`.

4. Also there are well-accepted notions which might as well be common across languages. Those native to the JVM all have natural generalizations to this level (e.g., element applied to a list or other collection, new applied to factory or mixin APIs). There are `as:` (conversion) and `operator:` (symbols which appear as non-name operations), and perhaps even `if:` and `for:`, although these probably fall down into language specifics. We need a notion of events for reactive or event-driven programming in the relevant languages.

5. The notion of imperative side effects factors tantalizingly across most operators (field, element, method call). This probably means we need a nomenclature syntax for inverting a getter name into a setter name, regularly. Hence the `set:` prefix, as in `set:field:f`, `set:method:f`.

6. Similarly, there are other modifiers that factor across some of the other constructs: Null handling notations `foo?.bar`, multiple-value notations `foo*.bar` or `foo(*bar)`, complex get/set patterns `foo[*bar] = baz` or `foo() += bar`. How are these extra conditions mixed into the basic selection of field/element/method? Are these extra notations mixed in as prefixes (`nullable:method:foo`) or as suffixes (`method:foo:nullable`)? Probably prefix-only, to avoid ambiguity, except for certain modifiers (like type or method arguments). Or is there a way to mix several `syntactic aspects` together on an equal footing? (Yes, something like XML attributes. See argument attributes below as an example.)

7. How do we leave open spaces for reified generics, if the JVM gets them, or if a language chooses to simulate them? This is another (complex) case of name modification. Adding bracket tokens, maybe we have `class:list:(:int:)`.

8. What other degrees of freedom are needed for structured names?

9. What is the "polite" human readable form for such names? (E.g., replace colons by spaces, replace empty tokens by a fixed hole token (like {_}) and further adjust spacing for legibility.) This point assumes we fail to reach our goal at point #1 above.

Argument annotations

Since a call site is all about stacked arguments and return values, each call site is governed by a signature or descriptor which specifies exactly which JVM-level types are pushed and/or popped at a call site. These types are not language-level types. Even for Java, they are "erased" types which omit certain statically checked generic type information.

So there is a sizeable set of argument annotations which do not affect the way the arguments would be stacked but are syntactically present at the call site, and are significant to function linkage or runtime typing.

Here are some:

A. Whether a transmitted argument is spread or not (as with Lisp &REST or Groovy `f(*x)`; affects calling sequence adaptation).

B. Whether a transmitted argument is a (reified) type parameter (as in Scala `f[x]` vs `f(x)`; not sure when/if they are reified, though).

C. Whether a transmitted argument is keyword-modified (and by what keyword).

D. If it matters, whether a transmitted argument is an appended block ({{f(x)

Unknown macro: {y}

}}).

E. Any additional language specific type or constraint information not present in the JVM-level type. (Example: nullability, array length, generic type information, structural or advisory typing.)

F. Whether two or more transmitted arguments are to be grouped logically as one. (E.g, type `complex(re,im:real)`.)

G. Whether a transmitted argument is to be hidden from the receiving method. (Perhaps it is an implicit argument to a parameterizable calling convention; can't think of an example. B above is a special case of this.)

Many of these annotations need to shake hands with corresponding definition-site parameter annotations.

Apparently we need some way of affixing such information to a method name, as `method:foo:(:...:):`. Parameter annotations would correspond positionally to stacked arguments.

```
call = myfunc(a1, a2, key: a3, otherkey: a4)
name = "method:myfunc:(:,:,:keyword=key:,:keyword=otherkey:):"
```

When presenting to polite human company, rely on ordinary (complex) conventions for presenting token-based expressions:

```
method function(_, _, keyword=key, keyword=otherkey)
```

Note that the latter friendly notation isn't necessarily accurate encoding, since a keyword might be an arbitrary string, including one containing a comma. (Probalby no language uses this particular degree of freedom, but it would be crazy to build symbol spelling restrictions into a new MOP.)

To transmit two attributes with one parameter, separate them by (say) a plus token:

```
method function(_, _, notnull + keyword=key, keyword=otherkey)
method:function:(:,:,:notnull:+:keyword=key:,:keyword=otherkey:)
```