The Saga of the Parametric VM

John Rose and the Project Vahalla team
June 2020 to March 2021 (ver-0.4)
References: PDF HTML slides

Introduction: Terms, Goals, Requirements

This document develops the design and use of *parametric* constants, methods, classes, and fields. The parametricity originates in the constant pool, and is threaded from there through the definitions and uses of parametric methods, classes, and fields. Any group of *co-parametric* constants and API points can be specialized coherently and efficiently.

Acknowledgements

Brian Goetz, Maurizio Cimadamore, Dan Smith, and Fred Parain have been indispensible in setting requirements for this design, and discussing various versions of it, along with the whole Vahalla design team. Dan and Fred originated the idea of type restrictions.

Design principles: JVM-centric, Factored, Predictable, Optimizable

This design attempts to focus on the actions of the JVM, pushing complexity (when possible) onto the language runtime via bootstrap methods and other upcalls. The upcalls factor out concerns which would be unwieldy to express in JVM specification and code.

Inside the JVM, we also attempt to factor in the new features so as to disturb already delicate parts of the JVM as little as possible. For example, the verifier is unchanged, as are inheritance and subtyping rules (as far as they are hardwired in the JVM). The structure of symbolic references is unchanged, except by addition, and that is in the constant pool node structure, not the syntax of class names or descriptors. In general, existing structures are enhanced by addition of side data, not by intrusive changes. The result is a design which is easier to prove safe and sane.

In validating the design, we have sometimes referred to Maurizio Cimadamore's thesis, which does a heroic job of pushing all of the Java 5 language down into the JVM, but does so by adjoining new structures to unchanged ones. Relative to that work, our goal here is to preserve the basic insights, of what new connections need to be "plumbed" to allow APIs to gain parametric aspects, and (contrariwise) which aspects of language implementations to keep out of the JVM, by referring them to upcalls. In the course of the exercise, we have found that parametric specializations look different at the JVM level from those at the source level, and have their own natural primitives and design space.

Because at the source level type parameters are part of static type information, and because the JVM encodes such information in the constant pool, we have integrated the necessary parametric "plumbing" into the constant pool, rather than adding either a new kind of dynamic argument (somewhere besides method arguments); nor have we needed to add a completely new kind of declaration (neither method arguments nor class constants).

The JVM has little to no interest in tracking type system proofs, beyond its required attention to the verifier and its own type system. The JVM has a compelling interest in tracking parametric information so as to build specialized code and data structures. Thus, the end result of successfully tracking of parametric constants and API points is the specialization and successful optimization of those constants, leading to tighter data and faster code.

By mixing parametrics into the constant pool we find they are exactly where we need them (next to symbolic references). As a bonus, ad hoc specialization transforms are easy to express using condy.

The shape of these parametric constants may look surprising. Language level type parameters are completely invisible. A caller may add at most one *linkage parameter* (a static value) to a symbolic reference to given API point, and the corresponding resolved declaration may specify a *specialization anchor*, which receives the linkage parameter and makes use of it to drive specialization logic. For any given API point, any and all specialization decisions are encapsulated within the class file that declares the API point and its specialization anchor. These decisions are permanently recorded in the caller's constant pool as resolved linkage state, but they may only be inspected by the class file declaring the API point.

In short, every API point use site can specify an optional linkage parameter, and (in aby single class file) any group of API point declaration sites can specify a specialization anchor to receive and act on linkage parameters.

One such parameter is enough "envelope", of course, for a language translation strategy to package up any amount of "mail", such as record-like tuples of reflective type variable bindings. The option to parameterize and specialize is applied broadly and evenly: Classes, interfaces, fields, methods, constructors, both definitions and use sites, are all equally and independently open to the presence of parametricity. Dynamic linking of separately compiled API points works just the same, but with an extra "piece of mail" added to every linkage event, and delivered wherever parametric instances are to be found, or parametric methods are called.

When a user of an API point supplies a linkage parameter along with a symbolic reference to the API point, the JVM's linkage resolution logic delivers the parameter value to the specialization anchor associated with the resolved API point declaration, in a particular class file. That anchor then makes a group of specialization decisions that include that API point, but may include other *co-parametric* API points and constants in the same class file. This set of decisions is private to the declaring class file, and the user can see only specialization details that the declaring class chooses to reveal.

All this is done with just two new constant pool types and two new class file attribute formats (for linkage parameters and specialization anchors), and the `Parametric` and `TypeRestriction` attributes, which can be attached to classes, fields, or methods.

We have not tried, yet, to simplify the work of compilers or designers of translation strategy. It seems premature to do this, since just getting the JVM parts right is plenty hard. Further prototyping is likely to show simple but helpful ways for the JVM to make compilers a little simpler–of course, short of moving the compiler logic into the JVM. We may add new constant pool item types (beyond the two we introduce here), if they are deeply useful. For now, folks, condy is your friend.

This design is organized so as to be optimizable using many pre-existing JVM techniques. It may also enable new techniques, such as smarter method customization. We double down on shaping "fast paths" (a common condition in the JVM, where speculation pays off) as well as "slow paths" (to be handled by deoptimization when possible) which cover less-important corner cases, such as support for raw types. The design does not, however, allow optimization to produce shifts in specified behavior. Specialization can never simply be disregarded by the JVM. Thus, a "dumb" JVM implementation and a highly optimizing one will process exactly the same linkage parameters and specialiaziton anchors, and so both will get the same results (if the latter waits up for the former to finish).

Our hope is to end up with a design which looks, more than other options, almost obvious in hindsight.

We will make further design observations as we go.

Terms: Parametric vs. Invariant, Specialized vs. Customized, Layout, etc.

Let's introduce the following terms with partial definitions. We will more fully define them later as needed in context.

- *Variant Type:* A type, in the language or VM, whose meaning can vary in different contexts.
  In Java source code, a variant type depends on a type variable and may vary in different applications of that type variable. A type depending on a wildcard is also variant. For example, if `T` is a variant type, then `List<T>` is, and vice versa, but `List<String>` is not variant. To represent types, the JVM uses reflective objects (such as `Class` objects), descriptor strings, and metadata items not directly accessible to Java code.

  In the JVM, variant types are not represented directly; they are *erased* at translation time to less informative types. (For example, `List<String>` is erased to `List` and `T` or `?` in `List<T>` or `List<?>` is erased to `Object`.) New parametric structures in the JVM also assist in tracking the identities and effects of variant types, although the JVM does not directly model them (beyond their erased forms). Similar observations can be made about generic fields, constructors and methods in Java source code.

- *Invariant Constant*: An item in a class file's constant pool which represents at most one value. Prior to this proposal, Java class files have only invariant constants. Note that a constant often describes a runtime type, and is often lazily resolved (with possible resolution failure). Note that invariant constants are used to translate erasures of variant types.

- *API Point*: A named class, interface, method, constructor, or field. Users (sometimes known as "callers") of API points refer to them via symbolic references, which are resolved to declarations (sometimes known as "callees") in specific class files. (A non-static API point has a distinguished argument called the "receiver" object, or in the case of a field, the "container" object.) Methods and fields have type descriptors which determine a static type. All API points, even ones which implement variant types, are defined in terms of invariant constants. API points which implement variant types have additional structure beyond their static names and types.

- *Specialization*: The management of distinct groups of constant resolutions and associated behaviors of multiple versions of a declared class, interface, field, method, or constant, as used by multiple clients. If a type or type member is specialized, its behaviors may be specialized. Specialization is implemented with a basic mechanism for tracking extra parametric constants associated with API points (affecting their instantiation, invocation, or access), plus runtime library code which shapes the tracked information into specialized classes, interfaces, fields, and methods. (Specialization does *not* transform or vary names, type descriptors, or bytecodes; these are always invariant.)

- *Specialization Anchor*: A new kind of class file constant pool item (tagged as `CONSTANT_SpecializationAnchor`) which embodies a single, coherent set of specialization decisions. Class file elements that depend on an anchor are specialized along with the anchor itself. (Others are invariant.) Thus, a single class file element can be specialized when and only when that element's anchor is specialized; conversely any specialization of the anchor determines a corresponding specialized behavior of the element. Specialization decisions embodied in an anchor may be accessible from specialized instances of a class or interface, or from specialized invocations of a method or constructor, as described below.

- *Parametric Constant*: An item in a class file constant pool which either is a specialization anchor itself, or else depends (directly or indirectly) on such an anchor. Its effective type, value, and/or behavior may vary across distinct specializations associated with the anchor. Many kinds of constants (including pre-existing kinds, such as `CONSTANT_Methodref` and `CONSTANT_Dynamic`) can be either parametric or invariant.

- *Parametric Class*: A class or interface (as defined by its class file) which is declared to depend on a specialization anchor, by means of a `Parametric` attribute that refers to the anchor. Specialized constants associated with this anchor are accessible from any instance of that class or interface. The types of fields in the class may be specialized. (Note: Following current usage as documented in `class-terminology-jls.html`, we will often use the combined phrase "class or interface" to describe an entity which is loaded form a class-file. Sometimes the plain term "class" will be used when misunderstanding seems unlikely.)

- *Parametric Method*: A method or constructor (in its class file) which is declared to depend on a specialization anchor, by means of a `Parametric` attribute that refers to the anchor. Specialized constants associated with this anchor are accessible within any invocation of that method. The effective type of the method may be specialized.

- *Parametric Field*: A field (in its class file) which is declared to depend on a specialization anchor, by means of a `Parametric` attribute that refers to the anchor. The effective type of the field may be specialized. The internal layout of the field may be optimized. (If the field is non-static, the enclosing class must be specialized on the same anchor.)

- *Type restriction*: A rule which applies to a field value, method return value, or method parameter, with the effect of blocking or excluding a specified subset of the values that are naturally available under the declared type of the value. (The rule may or may not refer to a subtype denotable by a type descriptor. See below.) Type restrictions are applied to API points to condition their behavior for better optimization. Type restrictions can be specialized.

- *API Point Name*, *API Point Reference*: An API point is used (or "called") by means of a resolved *API point reference* in the user's constant pool. This reference is often a symbolic constant of type `CONSTANT_Class`, `CONSTANT_Field`, `CONSTANT_Methodref`, or

`CONSTANT_InterfaceMethodref`. These symbolic constant types are called *API point names*.
As a new feature, any use of an API point name can also refer to a “decorated” API point reference (not just a symbolic name) that contains extra constant pool structure. The extra “decoration” requests a specialization of the API point. The meaning of symbolic references to API points is unchanged in this proposal, in the sense that an API point reference is symbolically resolved exactly as if the resolution were performed on an invariant API point name obtained by stripping out any “decoration”. See below.

- *Co-parametric*: Two elements (constants, API points) in a single class file are co-parametric when they directly depend on the same specialization anchor. (In a degenerate sense, invariant elements may also be viewed as mutually co-parametric. In this sense, the constants and API points of a class file form equivalence classes of co-parametric elements.) Elements that directly depend on a common anchor are interoperable under a single specialization of that anchor. It is typical for a parametric class to be co-parametric with some of its fields, all of its constructors, and some of its methods. Parametric elements which are not also co-parametric with their class may be called independently parametric (and may be co-parametric with one another). Two API points or constants in different class files are never co-parametric. In particular, specializations are not subject to inheritance; each level of a class hierarchy manages its own specializations.

- *Sub-parametric*: Occasionally, one specialization anchor may depend on another, specifically when a parametric method nests in an independently parametric class. In that case the class is not co-parametric with the method, but rather sub-parametric to the method, and (extending to the above equivalence classes), anything co-parametric with the class is sub-parametric to anything co-parametric with the method. If *C* is sub-parametric to *M*, then *M* can operate on *C* within a single specialization of *M*, because *M*’s specialization determines another specialization of *C*, and the anchor for *M* links to the anchor for *C*. (In a degenerate sense, invariant elements may also be viewed as sub-parametric to all other API points and constants. In this sense, there is a partial order between the previously mentioned equivalence classes. A set of co-parametric elements has natural access to elements sub-parametric to that set.)

- (*Variant*: Generally, the opposite of invariant, so not solely dependent on a static or once-resolved constant value. Can be used to describe something (constant, class, method, etc.) that is not invariant but rather parametric. Variance is an implementation requirement for a source code feature. Specifically, parametric API points and constants will be the recommended means to that goal, as opposed to variance obtained by other means, such as bytecode spinning or value-dependent types. In the JVM, the opposite of invariant is parametric, not variant.)

- *Preparation*: The phase of class linking which assigns memory resources to JVM states associated with a given class. In this document, preparation also contemplates the process of creating JVM states for resolvable constants. (These states are within the run-time constant pool, §5.1, as affected by the processes of resolution, §5.4.3.) At runtime, preparation is a prerequisite to assigning a fresh value to a resolvable constant (or assigning an initial value to a new static field). Once a constant (parametric or invariant) is prepared, it can then be resolved at most once. When a parametric constant is prepared, the run-time constant pool containing that constant expands by gaining new a resolution state for that constant. Before preparation, a constant is simply a static symbolic reference in a run-time constant pool, directly derived from a static structure in a class file. Immediately after preparation, any constant (invariant or parametric) will have a state of being unresolved; thereafter it can be either resolved to a value (either a loadable constant or an item of metadata) or resolved in error (with a recorded exception). Invariant constants are individually prepared “up front” during preparation of the containing class class. Parametric constants are prepared exactly when a new specialization is created. (The constants prepared are exactly those co-parametric with the anchor constant for the specialization being created.) Both invariant and parametric constants have the same rules for resolution, in common, as applied to their prepared states.

- *Resolution*: At runtime, the process of changing the state of a prepared, unresolved constant by permanently associating it with a loadable value, or an item of metadata, or a recorded exception. Invariant constants are resolved at most once, because they are prepared once. Parametric constants are (in general) resolved many times, because they are (in general) prepared many times.

- *Validation*: The process by which a linkage parameter proposed by the client of an API point is accepted by the specialization anchor of that same API point. A client cannot force specialization into an API point without validation. Each parametric API point has the “final say” on what values it uses, internally, to represent the variant semantics intended by the programmer and translation strategy. Validation thus defends encapsulation of API points, and supports separate compilation. In general, validation replaces a client-supplied value with an internal token called a *specialization anchor*. (As we shall see, this internal token is reified by a Java object of type `SpecializationAnchor`.) However, clients are allowed and encouraged to propose previously validated specialization anchors to API points, and the JVM efficiently accepts them without redundant revalidation.

- *Specialized class*: Generally, a class or interface which has been specialized somehow, with some sort of bookkeeping to record the decision. (The phrase *class specialization* refers either to the process of making specialized classs, or to a specialized class itself.) Specifically, in this proposal, a class (or interface) which depends on a specialization anchor, which has in fact been specialized. Subject to type restrictions or other variant behavior, a specialized class can be used instead of a normal, unspecialized class for at least some operations. The symbolic references used are the same in both cases. Two specializations of a class are the same only if they refer to the same specialization anchor. Differing specializations may exhibit differing behaviors or type restrictions.

- *Specialized method*: Generally, a method or constructor which has been specialized somehow, with some sort of bookkeeping to record the decision. (The phrase *method specialization* refers either to the process of making specialized methods, or to a specialized method itself.) Specifically, in this proposal, a method (or constructor) which depends on a specialization anchor, which has in fact been specialized. Subject to type restrictions or other variant behavior, a specialized method can be used instead of a normal, unspecialized method for at least some operations. The symbolic references used are the same in both cases.

Two specializations of a method are the same only if they refer to the same specialization anchor. Differing specializations may exhibit differing behaviors or type restrictions.

- *Class species* (or *interface species*): A user-visible type mirror for a specialized class (or interface) which can be used to manufacture instances (or subtypes), test instances, or make type restrictions. In general, class specializations may have private constants or API points are not relevant to the publicly visible species. Even more, it is possible that several specializations share a single species, so that a test for the species does not reveal internal distinctions made within the specializations. Still, in the simplest use cases for class specialization, each class species corresponds to a single unique specialization. In the current proposal, a specialized class's layout (field specializations) is linked to the species, and not to the specialization anchor (which can have additional variability to represent "private opinions").

- *Customization*: Generally, any process which enables the JVM to optimize an artifact that uses, accesses, or otherwise depends on a parametric API point or constant, by copying the artifact with the parametric API point or constant "hard coded" to a particular specialization. At the cost of extra versions of code and metadata (the customized artifacts) this can gains the performance benefits of invariance while preserving the flexibility of genericity. Customization can involve a mix of speculation, inference, profiling, and/or dynamic side channels. Although there are a number of occasions and implementations of customization, the common thread is extra "bookkeeping" to allow some variant type or value to be presented to its point of use without loss of necessary information. Classes, variables, and method bodies may be customized in various ways. The JVM may customize parametric classes with respect to their associated co-parametric constants and API points. Independently of specialization, the JVM may customize a supertype method to a receiver subtype. Also, the JVM may customize a method to a particular type or value of one or more arguments (either the receiver or not). None of these customizations are allowed to violate the semantics of the program being run, and they are all optional.

- *Layout*: Generally, the size and shape in memory of a data structure, notably a class or array instance. If a class has no parametric fields, its layout can be fully determined when the class file is loaded; this is called an *invariant layout*.

- *Customized layout*: A layout can potentially be customized if it has specialized fields that are constrained to hold only values consistent with particular types (or values or ranges of values).

- *Flat layout*: A layout is flat when it presents a set of variables without needless indirections or headers. If a variable is of an identity class type, it needs an indirection to keep track of identity and a header to allow subclasses to interoperate polymorphically. But for a variable of an primitive class type, any indirection to its fields (e.g., for boxing or buffering) is needless. (Likewise, if a class's contract does not mandate the preservation of object identity in some stored value, then an inlined representation of a value might be selected, even if it loses identity information.) When generics and primitives are combined, some kind of layout specialization is needed to achieve flat layouts. It is the responsibility of the runtime library to communicate to the JVM its intentions about which information to record about a class species, and whether or how to specialize the layout of the class. It is the responsibility of the JVM to customize layouts into flatter forms, if it can exploit the specialization information from the runtime, and if the effort is profitable. As will be seen, specialization anchors provide the necessary bookkeeping to track specialized layouts, so they can be customized when that is profitable.

- *Object code*: Machine instructions (optimized or not) which directly implement a method's actions. (Normally a JIT or AOT produces object code. In a certain way, a JVM bytecode interpreter can be viewed as object code for *all* methods.) *Variant object code* depends somehow (either statically or via dynamic computations) on one or more variant types or other constants. Typically, variant object code works with variables of variant types.

- *Specializable object code*: Object code is specializable if the variant types or values it uses are constrained to be specific types or values, so that the instruction sequences used to work with those types and values are then specializable to those types or values. Just as specialized layouts can eliminate useless indirections, specializable object code can omit useless boxing or buffering. Specializable object code can often devirtualize and inline many virtual calls on values of variant type, where unspecialized object code would make out-of-line calls through dispatch tables. It is the responsibility of the runtime library to communicate to the JVM its intentions about which information to record about a method species, and whether or how to specialize the code of the method. It is the responsibility of the JVM to customize object code and calling sequence to flatter forms, if it can exploit the specialization from the runtime, and if the effort is profitable. Specialization anchors provide the necessary bookkeeping to track specializable object code, so it can be customized when that is profitable.

- *Calling convention*: A convention shared between calling and called machine code for where (stack, heap, registers) to put arguments and return values during the call and return. Calling conventions are needed to coordinate separately compiled blocks of object code. In particular, a common calling convention must usually be agreed upon by all callers and implementors of a given virtual (or interface) method.

- *Customized calling convention*: A calling convention, appropriate only to a specialized caller and callee, where arguments or return values of variant type are represented more optimally according to the common constraints of the caller and callee. Specialization anchors provide the necessary bookkeeping to track calls to specializable methods, so the calls (and the methods) can be customized when that is profitable.

- *Flat calling convention*: A calling convention is flat when it presents a set of arguments and return values without needless indirections or headers. Primitive objects may be stored directly in stack memory or registers, not boxed or buffered in the heap. Specialization anchors provide the necessary bookkeeping to track specialized fields and their access, so their layout can be customized when that is profitable.

- *Default class specialization* (resp. *default method specialization*): If a class file defines some specialized behavior, then for certain "extra-special" purposes (such as wildcards or migration compatibility), the JVM will also define a standard "raw" layout and behavior as if it were

unspecialized.
This layout and behavior is not under user control. It minimizes bookkeeping by paying attention only to JVM type descriptors. (Recall that these encode the bounds of source language type variables, after erasure.)
The JVM keeps track of this extra case automatically, in addition to all other specializations, which are under user control. So is it "a unique and very special specialization"? Or is it "not a specialization at all"? Sometimes we think of it one way, and sometimes another.

Such species and their associated concepts are sometimes called "raw", always with "scare quotes", to emphasize a connection with a similar concept in the present Java language, that of a type or method which has "nothing to erase", because it already requires nothing more than the expressive capabilities of the present (non-parametric) JVM.

- *Default (or "raw") layout*: The layout of a default class species. It is invariant because it forgets about parametric type constants and remembers only the bounds. As such, it typically uses polymorphic indirections to uniformly represent field values of variant types, and therefore is not flat.

- *Default (or "raw") code*: The object code compiled for a default method species (or for some similar purpose) so as to handle all possible type arguments in the finite output of a (JIT or AOT) compilation task. It is invariant because it forgets parametric type constants and remembers only the bounds. (If it is used to execute parametric methods, it must rely on some hidden side-channel, managed by the JVM, to provide information about specialization decisions.) As such, default code typically uses polymorphic indirections (and/or data dependencies on specialization information) to uniformly represent field values of variant types, and therefore is not efficient. Default code is *also* a "one size fits all" fallback which works correctly (though not always efficiently) on customized layouts as well as default layouts.

- *Default (or "raw") calling convention*: The calling convention used by default method code. It is *also* a "once size fits all" fallback which can be used if a caller and callee fail to agree on a common specialized calling convention.

- *Reflective* use of default artifacts: When default code or a default calling convention is used as a fallback for a more desirable form of specialized code or calling convention, we say it is being used reflectively. Default calling conventions may include side channels for dynamically passed specialization information, and default code may use such side channels, even though no such side channels are present in today's Java generics. Thus, default code serves two purposes: First, to correctly execute in the presence of a default specialization (on default or "raw" instances); secondly, to correctly execute (perhaps with a performance penalty) in the presence of *any* specialization, by making data-dependent references to a runtime value reifying a current specializaiton anchor. An optimizing JVM can (if it wishes) separate these two concerns, in two (internal) versions of a method.

Goals and Requirements

Our overall goal is to support efficient generic programming, using Java's current generic constructs. Valhalla's primitive classes, with their characteristic firm guarantees of flattening in memory, add new requirements and challenges to generic programming in Java.

To maintain flattening of fields, arguments, and return values through generic code, we must enhance the current translation strategy to use techniques beyond erasure. The problem with erasure is that it requires pointer polymorphism, in order to retain type information about values of variant types, while still erasing the variant type down to its head or bound. But pointer polymorphism is incompatible with flattening, because it introduces extra indirections and/or object headers. Also, existing translation strategies fail to provide enough information to recover the original types (before erasure), so there is no amount of "extra optimization" that would take today's class files and reliably flatten generic data structures.

And flattening of instances is not the whole story. To avoid boxing or buffering along hot paths, there must also be (at least in some VM implementations) a coordinated flattening of calling sequences (when callers and callees agree on specializations) and also routine customization of method code, to keep primitive objects (both specialized and invariant) from falling out of registers, and to avoid expensive virtual calls.

A second overall goal is to design the JVM support for flattening and method customization so that it integrates smoothly with existing JVM functionality. It would be ineffective to create a new VM-within-a-VM just for customization, or to permanently hardwire today's exact theories of genericity in the Java language. Instead, as always, the quest is to find the correct primitives for the JVM to implement, primitives that are scoped to the natural operations and optimizations already present, or that cleanly and orthogonally extend those operations and optimizations. The result is likely to do both less and more than what a language-centric design effort would produce: Less, because some policy decisions (such as generic subtyping) might be delegated to the language (e.g., via bootstrap methods), and more, because some degrees of freedom (such as the "kinding" of parametric constants) might be simpler to leave open (e.g., parametric non-type values) even if the language has no immediate plans to use them.

FlatLayouts: Generic layouts can be flat
Specializations of generic classes for primitive classes will be easily available for use, and can (in some implementations) be reliably customized to use flat layouts containing those values. The size and type of fields of a class can thus vary from instance to instance.

For example:

- The non-empty payload of an `Optional<T>` can be stored directly in a field of the `Optional` instance, not indirectly via a pointer.
- The size of `Optional<InlineByte>` can be less than the size of `Optional<InlineDouble>`.
- A primitive record-like type `InlinePair<T,U>` can have varying sizes based on both `T` and `U`. (Note that this means one field must have a varying offset.)

Flat layouts are most useful when they are adopted from the first, even before the JIT has started compiling hot code. Flat layouts are not a JIT-time decision or optimization. Type variables must be tracked systematically in the interpreter as well as compiled code.

FlatCalls: Calling sequences can be flat
When one specialized method calls another, and the caller and callee agree on specializations, the calling sequence can (in some implementations) be reliably customized, so that boxing and buffering is avoided through the whole call chain.

For example:

- A flat `InlineOptional<InlineLong>` argument or return value can fit in two registers, one to contain the optional 64-bit payload, and the other to signal whether the payload is present.
- A primitive record-like type `InlinePair<T,U>` can be passed as an argument or return value in the union of registers and stack locations required to pass the two components individually. (There are the usual caveats about limited numbers of argument and return registers.)
- If an argument or return value is nullable, but non-`null` values can be flattened, the JVM can assign a special encoding to `null` to avoid using a physical reference. For example, a second register assigned to encode the presence or absence of a `InlineOptional<InlineLong>` value could be overloaded (with a third possible value) to additionally encode the presence of `null`.

For technical reasons, customized flat calling sequences sometimes cannot be computed lazily, waiting until after "hot spots" develop. This seems especially true in v-tables (type-sensitive dispatch tables). In such cases, decisions about flattening data structures and scalarizing method APIs must done "up front", before a JIT can run.

ScalarCode: Generic method code can scalarize
Specializations of generic methods to primitive classs will be easily available for invocation, and will have access to at least enough specialization information to (in some implementations) reliably produce and operate on scalarized instances of associated specialized generic types. If boxing or buffering of values is a performance hazard, there will be a way (for hot paths at least, in some implementations) to customize code enough to lift values out of boxes and into registers.

Unlike data structure layout and method APIs, the internal code of any single method can be optimized at any time (either early, or after a hot spot develops), and reoptimized at will.

RawSupport: Java "raw" types and methods are supported
Raw specializations of classes and methods are supported. Whatever bookkeeping is used to keep track of parametric constants can also record that some species intend for their type parameters to be unspecified. (Similarly, the erased and "wildcard" states, if different, are also supported, perhaps by different mechanisms.)

What's "raw"? At the source code level, "raw" refers to a use of a class or method which refuses to specify any type parameters, and instead expects that the class or method will behave consistently with the rules which predate Java 5 generics. Semantically, "raw" behaviors can be identified with the behaviors of Java API points after they have been compiled using erasure, and specifically with Java API points as observed through the Core Reflection APIs. Even if an object has specialized (non-raw) internals, its API points can be observed either reflectively or through "raw" symbolic references, from legacy code or from intentionally erased modern code.

RawInstancesUniversal: Raw class instances are always allowed
Any bytecode which is sensitive to class specialization, and which operates on an instance of a specialized class, will always accept either an instance of the class specialization proposed by the caller, *or else* provide a compatible fallback behavior when presented (instead) with an instance corresponding of the raw type (however that is represented).

This implies alternate paths for handling raw layouts, even in code which is optimized for specialized classes. Such alternate paths, if used, are likely to carry an extra cost.

If (as is proposed here) the raw type is represented by a distinguished "default specialization" supplied by the JVM, this requirement also implies a subtle distinction between the "raw species" as a narrow type (e.g., to impose on new instances), and as a universal "wild card" type (which is accepted everywhere).

RawMethodUniversal: Raw method calls are always allowed
Any bytecode which is sensitive to method specialization, and which invokes a specialized method, will *also* support a "raw" invocation mode which operates correctly on arguments of all specializations, and not just on those corresponding to a particular specialization requested by the caller.

Under such an invocation mode, the parametric method behaves as a "raw" species of itself. Raw execution will typically be slower than specialized execution because of the need to re-derive specialization information from arguments. It may also have incompatibilities with method code which expects to derive specialization information without the help of a "witness instance". In that latter case, the raw method species will supply a fallback behavior, such as creating additional instances of "raw" types, instead of parametric types.

ReflectiveSupport: Reflective access is supported
Species can be created, queried, instantiated, and invoked reflectively. Invocations and instantiations display the same "bytecode behavior" as if the call were not reflective but native in equivalent bytecode. (This implies that there are reflective API points which reify specialization anchors passed into and out of reflected APIs which are parametric, as well as reflective API points which present unspecialize "raw" bytecode behaviors.)

If an API point has a type restriction (e.g. of `Object` to `String` in the `get` method of `List<String>`), the restricted type can be queried reflectively.

Within a class file constant pool, there is some means for deriving all such reflective entities as loadable constants, relative to resolved API point references in the same constant pool. (E.g. `ldc` of an appropriate species reference.)

Reflective processing may subsume the implementation technique of having a fallback for "slow paths" that occasionally branch out from failed speculations, such as when code optimized for an flat layout containing inline values suddenly encounters a raw layout, containing buffered inline values.

IndependentSpecialization: Specialization is independent at each API point
A symbolic reference to a variant API point can meaningfully resolve whether or not the caller and callee have been compiled consistently. Inconsistent specializations can be recovered from if the translation strategy defines a consistent net semantics. The extra structures created by specialization are local to each class file, and require no fixed invariants between class files. This is true for all class file relations mediated by dynamic linking, including for callers and callees, and for subtypes and supertypes. The only way for two API points to be co-parametric is for them to be declared in the same class file.

This is a VM-oriented "right-sizing" of the requirement that legacy clients be able to operate compatibly on API points which have been upgraded to be parametric. Also, it doubles down on the primacy of the existing architecture of API points, as classes, interfaces, fields, and methods, and avoids surfacing new fundamental API points for (e.g.) inheritable type variables. Such new API points can created efficiently by translation strategy which mandates new synthetic methods, but they are not a direct burden for the JVM.

As an implication of this, the client of an API point is always free to *propose* a type parameter (or other specialization request), but it cannot *impose* any such condition on an API point that chooses not to specialize. (Runtime diagnostics for failed specialization requests are a matter for further prototyping TBD. It seems they can be added into runtime support code, at the option of the translation strategy.)

Even within a single class file, most API points can be separately and independently specialized. Of course, co-parametric groups of API points will typically be generated.

EncapsulatedSpecialization: Specialization decisions are private
The full information about a specialized API point is not exposed to any client of that API point; it is encapsulated within the class file that declares the API point. The class file is in control of how much information is exposed to clients of the class file. This control is expressed using existing mechanisms of access control, which implies that translation strategies may need to create synthetic API points (e.g., `public` or `protected` methods) to selectively expose specialization information that is otherwise encapsulated.

The encapsulated information will include the `SpecializationAnchor` object described below, which manages constant pool states and type restrictions, and is (usually) created in response to a bootstrap method upcall. (The upcall is, like other similar bootstrap method calls, given full access to the internals of the relevant class file, via a `Lookup` object.) Specialized type information for any given API point (if any) is available to any client who can access the same API point, since this information is necessary to provide to any external client of that API point. The number and nature of particular specialization decisions (which are reified by various `SpecializationAnchor` objects) are not accessible to clients unless the specializing class chooses to expose them somehow, or a reflective API exposes them.

ClassVariance: A method can be specialized along with its enclosing class
There must be an efficient translation of methods which make non-trivial use of type parameters from their declaring class or interface. In their class file, such methods will be co-parametric with the class or interface.

```
interface VariantType<T> {
   void cospecialized(T arg);
}
```

MethodVariance: A method can be specialized independently of its enclosing class
There must be an efficient translation of methods which make non-trivial use of type parameters declared independently of their declaring class or interface. In their class file, such methods will independently parametric of (not co-parametric with) the class or interface.

```
interface InvariantType {
   <U> void specialized(U arg);
}
```

BiVariance: A method can be specialized to both possible scopes
There must be an efficient translation of methods which make non-trivial use of type parameters declared *both* in their declaring class or interface *and* independently of it. In their class file, such bi-variant methods will be co-parametric with the class or interface, and will *also* have independent specialization.

```
interface VariantType<T> {
   <U> void bispecialized(T arg1, U arg2);
}
```

This requirement will be technically more difficult to fulfill than the previous two. However, careful implementation of the first two requirements makes this one easier to implement also. The key is finding the right primitives for the first two, so that the third becomes a new combination of existing primitives, rather than a new primitive.

No other form of multiple specialization is required, as long as a single VM-level parameter can represent a whole "pack" of formal type variables. This is because the only way that API elements can nest, in today's class file format, is if the outer element is a type and the inner one is one of its members. This requirement supports the maximum possible amount of specialization nesting, in today's class file format.

Volume I: A Parametric Classfile (JVMS-4)

Our starting point is to extend the existing constant pool structure to carry the existing variety of constants with a new twist: An entry in the constant pool (representing a type, another API point, or a constant value) can be declared parametric. The value (after resolution) of a parametric constant can be specialized (with resolution occurring once per specialization). Thus a single entry in a constant pool can resolve to different constant values for different specializations of a single class or method.

Pulling on this string leads us to interesting questions: How are specialized values declared and (for each specialization) defined? Which variation of a constant is in force at any given point? How are multiple specializations created, propagated, and prevented from conflicting during JVM execution? Most importantly, are the proposed JVM mechanisms simple enough to engineer well, yet powerful enough to support a useful range of new language features?

Parametric Constants

The constant pool is enhanced with two new structures, which also interrelate with many of the existing `class` file structures.

`CONSTANT_SpecializationAnchor`

The new `CONSTANT_SpecializationAnchor_info` structure is used to declare a distinct degree of freedom of parametricity for specializable API points declared in the same class file. It has this form:

```
CONSTANT_SpecializationAnchor_info {
    u1 tag;  // CONSTANT_SpecializationAnchor =
21
    u1 anchor_kind;  //
PARAM_{Class,Method{Only,AndClass}} = {1,2,3}
    u2 bootstrap_method_attr_index;
}
```

In diagrams and informal narrative, `CONSTANT_SpecializationAnchor` may be abbreviated as `CONSTANT_Anchor` or `C_Anchor`.

The items of the `CONSTANT_SpecializationAnchor_info` structure are as follows:

- The `tag` item has the value `CONSTANT_SpecializationAnchor` (21).

- The value of the `anchor_kind` item must be in the range 1..3. The value denotes the *kind* of this anchor, which characterizes the way constants derived from this constant may vary relative to other entities in this `class` file.

Note: The same information could be encoded by replacing the `anchor_kind` field with a `parent_anchor` field that either contains a null index (i.e., zero) or points to the enclosing class-kinded anchor. This would be more appropriate for a scalably nesting multi-class file format; we leave it on the shelf for now.

- The value of the `bootstrap_method_attr_index` item must be a valid index into the `bootstrap_methods` array of the bootstrap method table (§4.7.23) of this `class` file.

There are three kinds of parametricity:

- If the value of the `anchor_kind` item is 1 (`PARAM_Class`), the specialization anchor declares a degree of freedom which applies to the current class, as a whole. Any such `PARAM_Class` anchor, if it exists, must be unique in this `class` file, and must also be explicitly mentioned by the `Parametric` attribute of this class.

- If the value of the `anchor_kind` item is 2 (`PARAM_MethodOnly`), the specialization anchor declares a degree of freedom which applies to a set of methods of the current class. Each `PARAM_MethodOnly` anchor varies independently from all other anchors.

- If the value of the `anchor_kind` item is 3 (`PARAM_MethodAndClass`), the specialization anchor declares a degree of freedom which applies to a set of methods of the current class. Each specialization of a `PARAM_MethodAndClass` anchor is defined as dependent on another specialization of the `PARAM_Class` anchor in the same `class` file. If there are any `PARAM_MethodAndClass` anchors in a `class` file, there must also be a (single) `PARAM_Class` anchor also.

A `CONSTANT_SpecializationAnchor` constant is a (new sort of) loadable constant (§5.1). The resolved value of this constant is a mirror to a set of specialization decisions, also called a `SpecializationAnchor` (§4.1).

Note that, like `CONSTANT_Dynamic_info` and `CONSTANT_InvokeDynamic_info` structures, a `CONSTANT_SpecializationAnchor_info` structure refers to a bootstrap specifier (i.e., a method plus a static argument list). Unlike those other constants, a `CONSTANT_SpecializationAnchor_info` has no additional symbolic data in the form of a `CONSTANT_NameAndType_info` structure. As will be seen later, when the JVM invokes a bootstrap method for a specialization anchor, the bootstrap method calling sequence will be different than in the case of those other constants.

As will be seen later, `CONSTANT_SpecializationAnchor_info` structures can potentially be referenced by other constants in the same `class` file, as well as the current class and any of its fields or methods. Any `class` file structure (constant, class, method, or field) which depends on an anchor becomes parametric, and obtains special processing from the JVM. Any parametric structure which is variant depends directly on a single anchor, which determines the circumstances under which the variations take effect. Within broad limits, any two API points or constants in the same class file can be co-parametric.

It is envisioned that, in most cases, each distinctly scoped group of type variables in Java source code will correspond to a unique `CONSTANT_SpecializationAnchor` constant. However, if a number of generic methods in one classfile have identical type parameter declarations, it could be valuable for a translator to assign a single `CONSTANT_SpecializationAnchor` constant to represent the parametricity of all the identically declared generic methods, in common. If the methods (as seems likely) were to work in concert in a larger call tree, that call tree could link itself with fewer validation steps, since the methods working in concert would be working from a common `CONSTANT_SpecializationAnchor` constant.

`CONSTANT_SpecializationLinkage`

The new `CONSTANT_SpecializationLinkage_info` structure may be used to add parametric information to a symbolic reference to a class, interface, method, or field. As such it has two components, an invariant symbolic reference (§5.1), and a *proposed linkage value* to use along with the reference. It has this form:

```
CONSTANT_SpecializationLinkage_info {
    u1 tag;  //
JVM_CONSTANT_SpecializationLinkage = 22
    u2 parameter_index;
    u2 reference_index;
}
```

In diagrams and informal narrative, `CONSTANT_SpecializationLinkage` may be abbreviated as `CONSTANT_Linkage` or `C_Linkage`.

The items of the `CONSTANT_SpecializationLinkage_info` structure are as follows:

- The `tag` item has the value `CONSTANT_SpecializationLinkage` (22).

- The value of the `parameter_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a loadable constant (§5.1). (It will be proposed as a linkage parameter value for the associated API point, and validated produce a specialization anchor for that API point.)

- The value of the `reference_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at index must be an API point name, that is, a `CONSTANT_Class_info`, `CONSTANT_Methodref_info`, `CONSTANT_InterfaceMethodref_info`, or `CONSTANT_Fieldref_info`.
  Note: `CONSTANT_InvokeDynamic_info` and `CONSTANT_Dynamic_info` are never wrapped in `CONSTANT_SpecializationLinkage` constants, because they do not refer directly to API points. They can easily propose specializations via their static arguments, either directly or indirectly via previously validated `CONSTANT_SpecializationLinkage` constants.

After successful resolution, a `CONSTANT_SpecializationLinkage` constant will permanently refer both to an API point (class, interface, method, or field), with an additional specialization anchor. Both components (API point metadata pointer and specialization) will be permanently available (locally) for all further uses of that API point. In general, when that API point is used, that (local) reference value will be bound to a corresponding CONSTANT_SpecializationAnchor in the (remote) definition of that API point.

Several CONSTANT_SpecializationLinkage constants may propose distinct linkage parameter values to the same API point, such as for List<InlineInt> vs. List<InlineDouble>. Bytecodes may select a specific linkage parameter value by referring to the appropriate CONSTANT_SpecializationLinkage constant for that value. Conversely, several CONSTANT_SpecializationLinkage constants may apply the same linkage parameter value to distinct API points, such as for List<InlineInt>.get and List<InlineInt>.set. Bytecodes using a set of such linkage parameter constants can expect to use those various API points (presumably co-parametric) with a single consistent setting of the anchor.

A CONSTANT_SpecializationLinkage constant is a (new sort of) loadable constant (§5.1). When loaded, its resolved value denotes a value chosen by the translation strategy during the execution of a relevant bootstrap method call during resolution of the API point.

The translation strategy is free to define this value according to its own conventions. The value could be the SpecializationAnchor that underlies the specialized API point, or it could be a reflective species object, or it could be a representation of type arguments, or it could be an associated type restriction record. The JVM makes no policy about this value.

The JVM *does* secretly record any SpecializationAnchor associated with the resolution of a specialized API point, so that it can be present at all uses of that API point. There is no guaranteed way for client code to get access to the SpecializationAnchor reference, even though it is sitting in resolution state of the client's constant pool.

In a previous version of this proposal, the constant value of a CONSTANT_SpecializationLinkage constant gave more information: It was defined as identical to the corresponding specialization anchor of the API point resolved through the linkage constant. This semantics is more powerful, but also is thought to "leak" too much information from the implementation of the API point. Instances of SpecializationAnchor object can be shared, of course, via explicit API points created by translation strategies, but the present design protects the encapsulation of SpecializationAnchor objects by default.

If the bytecode behavior of a parametric field or method reference is desired (as a loadable constant), wrap the appropriate CONSTANT_SpecializationLinkage constant in a CONSTANT_MethodHandle constant. The MethodType of the resolved MethodHandle constant will reflect type restrictions. The unrestricted type is recoverable via MethodHandleInfo::getMethodType.

If validation fails, the resulting exception will become the resolution state of the constant. Just as in the case of a failed symbolic resolution, a failed validation can prevent bytecodes which use a specialization from completing normally. If the failing constant is parametric, then (consistently with the distinction of resolution states of distinct specializations) some specializations can fail while others succeed. In all cases, for any given CONSTANT_SpecializationLinkage constant, there is just one outcome per prepared resolution state.

In summary, assuming successful resolution of both components of a CONSTANT_SpecializationLinkage constant:

- If the reference_index refers to a method, and that method is subsequently invoked (via the same CONSTANT_SpecializationLinkage constant), the JVM will pass the resolved specialization anchor as an extra hidden argument into that method's call frame.

- If the reference_index refers to a field, and that field is subsequently accessed (via the same CONSTANT_SpecializationLinkage constant), the JVM uses the resolved class specialization anchor to locate that field. The instance containing the field (if it is non-static) is dynamically checked to ensure that its class specialization is consistent with the one expected by the field reference constant.
  Field polymorphism will be allowed in some cases such as when the reference uses the default specialization anchor (a "raw" reference). In most cases apart from wildcards, there will be an expected field type that will be exactly fulfilled (modulo a slow path). This code shape is more analogous to the invokeExact call on method handles than generic invoke.

- If the reference_index refers to a class or interface, and that type is subsequently resolved (via the same CONSTANT_SpecializationLinkage constant), the JVM records that class's specialization anchor in the resolution state, for use in returning a species (via ldc) or performing further linkage to members of that species.

We say the API point is "remote" to emphasize that it may be declared in an arbitrary class file, which in general is separately compiled independently of the "local" class file that is performing the linkage operations. In these terms, which are client-centric, CONSTANT_SpecializationAnchor constants are remote and CONSTANT_SpecializationLinkage constants are local. Of course, a class file may also resolve symbolic references to API points declared in the same class file. (In fact this is how a class gets access to its own private members.)

Just as a plain symbolic reference (of any sort) mediates access to an API point agreed upon by two class files, a CONSTANT_SpecializationLinkage constant mediates access to an API point (in exactly the same way), with the independent addition of a linkage parameter specified by the caller's class file, validated by the callee's class file, and recorded as a specialization anchor for the callee.

The new concept of *API point reference* extends the pre-existing concept of a symbolic reference. An API point reference can be any of the following:

- An invariant constant may represent the name of an API point, i.e., a class, interface, field, or method. (This has been true in all versions of the JVM.)

- A CONSTANT_Class may be wrapped in a CONSTANT_SpecializationLinkage item, thus embodying a parametric API point reference, to a "species" of a class or interface.

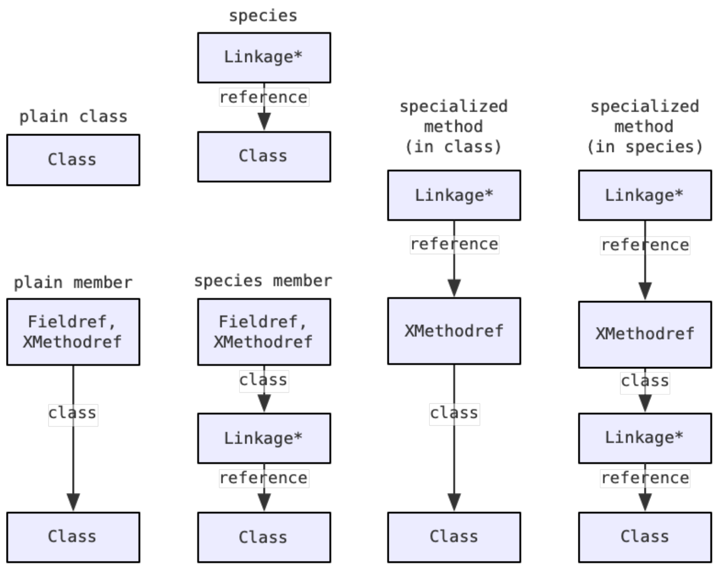
- A CONSTANT_Fieldref, CONSTANT_Methodref, or CONSTANT_InterfaceMethodref may be wrapped in a CONSTANT_SpecializationLinkage item, thus embodying a parametric API point reference, to a specialized field or method (of a class or interface, or of a species of class or interface).

- A `CONSTANT_Fieldref`, `CONSTANT_Methodref`, or `CONSTANT_InterfaceMethodref` may refers to its internal `CONSTANT_Class` item via an intervening `CONSTANT_SpecializationLinkage` item, thus embodying an API point reference within a species of the class containing the field or method.

Note that a reference to a field or method can be doubly parametric, when its internal `CONSTANT_Class` item is wrapped in an internal `CONSTANT_SpecializationLinkage` item, and the field or method reference is also wrapped, as a whole, in another `CONSTANT_SpecializationLinkage` item. Such a doubly parametric API reference typically resolves to a bi-variant member of a species.

All of these symbolic references, both invariant (purely symbolic) and parametric, are *API point references*. Their various configurations are summarized in Diagram 4.4-F(b).

`Diagram 4.4-F(b). "API Point References", all configurations`
`(These are also "API Point Names", except Linkage constants.)`



Dependencies between constants
There are new relations between certain existing constants and the new constants. By depending (directly or indirectly) on some `CONSTANT_SpecializationAnchor` constant, some existing constants can become parametric.

A constant structure *A* depends directly on another constant structure *B* if and only if one of the following circumstances is true:

- *A* contains an index referring to *B*.
- *A* contains an index referring to an entry *E* in the bootstrap method table (§4.7.23) of this `class` file, and one of the static arguments of *E* refers to *B*.
- *A* is a `CONSTANT_SpecializationAnchor_info` of kind `PARAM_MethodAndClass` and *B* is the corresponding `CONSTANT_SpecializationAnchor_info` of kind `PARAM_Class` (which must exist and be unique).

The transitive closure of direct dependency is simply called *dependency*, and the condition of dependency without direct dependency is called *indirect dependency*.

Thus, a constant structure *A* depends indirectly on a constant structure *C* if and only *A* does not dependent directly on *C*, but one or both of the following circumstances is true:

- *A* depends directly on some *B* which depends directly on *C*.
- *A* depends directly on some *B* which depends indirectly on *C*.

We say simply that a constant *A* depends on a constant *B* if *A* either depends directly or depends indirectly on *B*.

Dependency is a static, syntactic relation between constant structures in the constant pool of a `class` file.

Dependency can be circular, although this requires special care to avoid infinite regression during resolution. A constant *A* can depend on itself if and only if it depends directly on itself, or else it depends directly on another constant *B* that in turn depends on *A*.

The following structural constraints are enforced on dependencies between constants within the constant pool of a `class` file:

- No `CONSTANT_SpecializationAnchor` may depend on itself. (…Because it must be possible to resolve its arguments before it is bootstrapped.)
- If there is an anchor of kind `PARAM_Class`, it is unique in the constant pool of the current `class` file. (…Because there is exactly one class per file, at least at present, and because class specializations don't nest inside any other specializations.)
- If any constant *A* depends on some anchor *R* of kind `PARAM_MethodOnly`, then *A* depends on no other anchor of any kind. (…Because method-only specializations do not nest inside any other specializations.)
- If a constant depends on some anchor of kind `PARAM_MethodAndClass`, it depends on the anchor of kind `PARAM_Class`, and no other anchor. (…Because method-and-class specializations nest only in class specializations.)

A constant *A* is said to be parametric (or sometimes "variant" as opposed to "invariant") when it depends on a `CONSTANT_SpecializationAnchor` constant *R*. A parametric (or variant) constant *A* is said to be "parametric over" (or "variant over") an anchor *R*, if it depends on *R*. (Briefly, we can say "*A* is *R*-variant".) Also, a `CONSTANT_SpecializationAnchor` constant is said to be parametric over itself, even though it does not depend on itself.

Thus, parametricity (over some *R*) originates in an anchor (*R*) and is passed to all constants which depend on it. Also, any constant is therefore in one of these categories:

- It is invariant, neither an anchor, nor depending on any anchor either directly or indirectly. (This is the status of all constants in any class file that lacks specialization anchors.)
- It is parametric over an anchor (the unique one) of kind `PARAM_Class`, but no other anchor. Such a constant may

be called “class-variant”, relative to the class or interface defined by the `class` file.

- It is parametric over a single anchor of kind `PARAM_MethodOnly`. Such a constant may be called “method-variant”, in every method which refers to it via its `Parametric` attribute (§4.6).

- It is parametric over an anchor of kind `PARAM_MethodAndClass`, as well as over the anchor of kind `PARAM_Class`. Such a constant may be called “doubly-variant” or “bi-variant”, in every method which refers to it via its `Parametric` attribute (§4.6). For such a constant we say that its `PARAM_MethodAndClass` anchor is the “inner” or “more specific” anchor and the `PARAM_Class` it depends on is the “outer” or “less specific” anchor".

If two constants are variant in common over exactly one anchor, we say they are *co-parametric* with each other and with that anchor. If two constants are bi-variant in common over exactly the same two anchors, we also say they are co-parametric with each other and with the inner anchor. If we also say that all invariant constants are mutually co-parametric, then the relation between co-parametric constants divides the constant pool into 1+*N* equivalence classes, where *N* is the number of anchors in the class file.

We may also say that class-variant constants (including the class-variant anchor itself) are *sub-parametric* to all bi-variant constants (including anchors) in the same class file.

The resolution of any given `CONSTANT_SpecializationAnchor` constant will be seen (§4.X) to make use of the bootstrap method and static arguments. The restrictions on constant dependencies listed above imply that the bootstrap method and its static arguments must be invariant, unless an anchor bi-variant, in which case any of its dependencies may also be class-variant (sub-parametric to the bi-variant anchor).

An invariant constant has at most one resolved value, globally. Though it does not depend on any specialization anchor, an invariant constant may make use of parametricity mechanisms in the JVM. For example, if `ArrayList` were a class, and `ArrayList<InlineInt>` were a species of that class, a constant referring to the latter species would be invariant.

The following new direct dependencies are allowed between existing constants and the new ones:

- Any constant that refers to a `CONSTANT_Class` constant can instead refer to a `CONSTANT_SpecializationLinkage` constant which wraps an equivalent `CONSTANT_Class` constant. In particular, the `class_index` field of a `CONSTANT_Methodref`, `CONSTANT_InterfaceMethodref`, or `CONSTANT_Fieldref` may refer to a `CONSTANT_SpecializationLinkage` constant that in turn refers to a `CONSTANT_Class` constant via its `reference_index`.

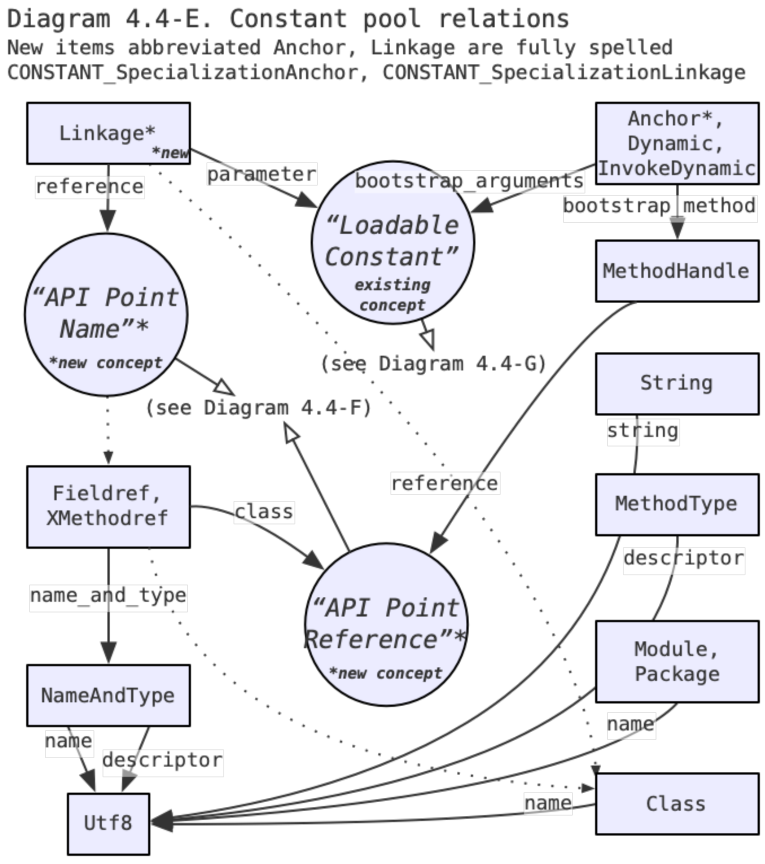
- Any constant that refers to a bootstrap method (`CONSTANT_InvokeDynamic`, `CONSTANT_Dynamic`, or `CONSTANT_SpecializationAnchor`) may depend directly on a `CONSTANT_SpecializationAnchor` constant or a `CONSTANT_SpecializationLinkage` constant as one of its static arguments, because both of those new constants are in the pre-existing category of loadable constants.

- A `CONSTANT_MethodHandle` constant may depend directly to a `CONSTANT_SpecializationLinkage` constant. The behavior of the resulting method handle will be derived from the corresponding bytecode behavior (§5.4.3.5), as modified by the specialization, which therefore must be incorporated into the resolved method handle.

The meaning of a `CONSTANT_SpecializationLinkage` constant depends on context. When used directly by a bytecode instruction to access an API point, it will denote the ordered pair of *both* a symbolic reference to an API point *and* a linkage parameter to validate for that API point. But a `CONSTANT_SpecializationLinkage` constant used as a loadable constant (via `ldc` or as a static argument), if it wraps a `CONSTANT_Class`, resolves simply to a species mirror for a specialization of that class. Other linkage constants (not wrapping `CONSTANT_Class` items) have no loadable value at all.

A `CONSTANT_MethodHandle` constant that refers to a `CONSTANT_SpecializationLinkage` constant will capture both components (API point and specialization anchor), in the form of an associated *specialized bytecode behavior* that depends both on the API point and on its specialization.

The relations between constant pool constants (both old and new) are summarized in Diagram 4.4-E.



Parametric API points and the `Parametric` attribute

The new `Parametric` attribute is a fixed-length attribute in the `attributes` table of a `ClassFile`, `field_info`, or `method_info` structure (§4.1, §4.5, §4.6).

Its purpose is to mark an API point (a class, interface, method, or field) as parametric and therefore specializable (variant) with respect to an indicated anchor constant.

The effect of this attribute is granular and independent for each API point. Any API point which lacks a `Parametric` attribute will be invariant and not subject to specialization. In particular, fields and methods do *not* implicitly partake of variance (of kind `PARAM_Class`) from their enclosing class or interface. For a field or method to be co-parametric with (or bi-variant over) the enclosing class, its `field_info` or `method_info` structure must contain a separate `Parametric` attribute selecting the anchor of the class (or a `PARAM_MethodAndClass` anchor, in the bi-variant case). A method whose `Parametric` attribute selects an anchor of kind `PARAM_MethodOnly` is *not* co-parametric with its enclosing class. Because specialization requires extra "bookkeeping" in the JVM, we never make fields or methods parametric by default, but rather require that parametricity is opted into by each API point.

The `Parametric` attribute has the following format:

```
Parametric_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 anchor_index;
}
```

The items of the `Parametric_attribute` structure are as follows:

- The value of the `attribute_name_index` indicates the string "`Parametric`".
- The value of the `attribute_length` item must be two (2).
- The value of the `anchor_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_SpecializationAnchor_info` structure (§4.4.X) denoting a specialization anchor *R* and representing the parametricity (i.e., variance) of the corresponding class, interface, field, or method.

A API point (class, interface, field, or method) is "parametric" (or informally "variant") if and only if the structure which declares it has a `Parametric` attribute. A parametric API point is "directly parametric over" the specialization anchor *R* indicated by the index stored in its `Parametric` attribute. A parametric API point is "indirectly parametric over" a `PARAM_Class` anchor *R* if it is directly parametric over a `PARAM_MethodAndClass` anchor *Q*, and it is simply "parametric over" an anchor *R* if it is either directly or indirectly parametric over *R*.

A class or interface may only be parametric over the (unique) specialization anchor of kind `PARAM_Class` in the same `class` file.

A non-static field may only be parametric over the same anchor as its enclosing class or interface (which thus must be of kind `PARAM_Class`).

Since a non-static field is part of the layout of its enclosing class, it cannot vary independently of the class itself (barring heroic hidden indirections).

Parametric static fields are TBD.

It seems likely that parametric static fields will be useful, and that their states can be conveniently implemented alongside their split constant pool states. Perhaps they can be stored inside relevant `SpecializationAnchor` mirror objects, just as class statics are stored in `Class` mirror objects. But there is no plausible language model for them yet. One problem is that `<clinit>` pseudo-methods cannot be made parametric in any useful way.

A method may be parametric over any kind of specialization anchor.

All API points that are directly parametric over some specialization anchor *R* are said to be co-parametric with each other. They are also said to be co-parametric with constants that are co-parametric with the same anchor *R*.

Loosely speaking, we are extending the dependency relation between constants to include API points as well, by defining that an API point depends directly on the constant referred to by its `Parametric` attribute.

As a general principle, any definition of a parametric API point (class, interface, method, field) is interpreted in the context of the indicated specialization anchor. Also, any parametric API point can be called (invoked or accessed) with a caller-proposed value that selects a specialization. Because specialization anchors are internal to each `class` file, and are not named directly from outside, the presentation of an anchor from a use to a definition is always in the context of some named API point. Thus, no additional naming mechanism is required to negotiate the mapping of linkage parameters to specializations.

This is in contrast to other systems, where an explicit parametric type system is built into the descriptors used by the managed runtime to select API points. Such an explicit type system had better be perfect, because it is much more difficult to evolve than a system of dynamic checks.

For compatibility and convenience, a caller is always permitted to omit a linkage parameter value. The callee is specified to use an internally generated default specialization anchor, which is set up when the API point's class is prepared. Also, callers can propose linkage parameters for callees which (after link resolution) turn out to be declared as invariant. (The treatment of such unused parameters is TBD. Perhaps they will be quietly ignored; perhaps there will be a diagnostic "hook".)

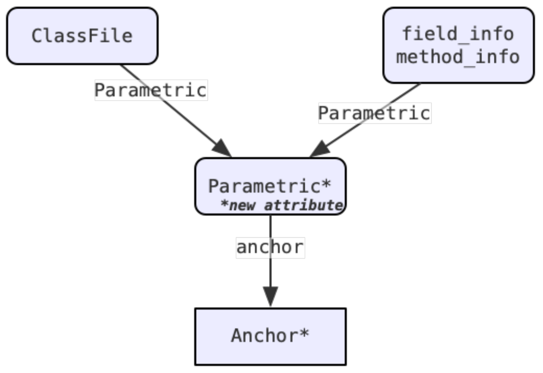
There is no direct mechanism for acquiring the *specialized type* of a parametric class member (field or method), although such data is surely useful. It can be acquired simply enough via a dynamic constant, which first computes a `CONSTANT_MethodHandle` constant for the specialized API point, and then extracts the `MethodHandle.type` property of the resulting bytecode behavior. More direct mechanisms may be created as needed. Translation strategies can also supply this information via the assigned loadable constant values of resolved `CONSTANT_SpecializationLinkage` constants.

There is no direct mechanism for acquiring a reflective class or interface *species*, although (again) such data is surely useful. Translation strategies can supply this information via the assigned loadable constant values of resolved `CONSTANT_SpecializationLinkage` constants. It is recommended (though not strictly required) that the resolved constant value of a `CONSTANT_SpecializationLinkage` constant which wraps a `CONSTANT_Class` constant should resolve to a species object that reflects the resolved specialization of the class.

To cover the previous two use cases (reifying specialized field and method types, and type species), a direct mechanism for extracting the type of an API point could be supported by a third new constant pool type, `CONSTANT_Species`, which extracts the type information more directly. Such a feature seems "nice to have", so we'll reserve it as a possible support for translation strategy. Probably there are more such "nice to have" features, which will be discovered as we prototype our translation strategies. For now, we will delegate the burden of organizing such information onto the translation strategies.

The relations of the `Parametric` attribute with other class file structures, including its reference to a `CONSTANT_SpecializableAnchor` item in the constant pool, are summarized in Diagram 4.7-D(a).

Diagram 4.7-D(a). Parametric attribute relations



Type-restricted methods and fields and the `TypeRestriction` attribute

The new `TypeRestriction` attribute is a fixed-length attribute in the `attributes` table of `field_info` or `method_info` structure (§4.5, §4.6).

Its purpose is to mark a field, method parameter, and/or method return value as (possibly) excluding values normally permitted by the type or types denoted by the type descriptor of the field or method.

A type restriction cannot add new values to the type of a field or method; it can only exclude values. A type restriction may be *ineffectual*, in that it excludes *no values* from any type denoted by the field's or method's type descriptor. A type restriction may be *unpassable* by excluding *all values* from one of the types denoted by the type descriptor; this makes a field or method impossible to use, causing an exception to be thrown in the excluded circumstances. A *trivializing* type restriction may make a field, method parameter, and/or method return type trivial (or "unitary") by excluding all *but one value* (the type's default value) from the corresponding type. A *null-excluding* type restriction may make a field, method parameter, and/or method return type *non-nullable* by excluding the value `null` from a corresponding reference type.

The preceding paragraph is a provisional account of functional requirements for type restriction objects. There is, in fact, no agreed-upon design yet for a type restriction API or implementation.

Prototypers should assume, for now, that the following items will be acceptable as type restrictions: (a) class mirrors, (b) species mirrors, (c) array type mirrors, and possibly (d) primitive mirrors (such as `int.class`) or (e) special tokens for trivializing and/or unpassable restrictions or (f) a token which wraps a reference type and declares it non-nullable. Perhaps all those will eventually implement some common protocol.

The `TypeRestriction` attribute has the following format:

```
TypeRestriction_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 restrictions_count;
    u2 restrictions[restrictions_count];
}
```
The items of the `TypeRestriction_attribute` structure are as follows:

- The value of the `attribute_name_index` indicates the string "`TypeRestriction`".
- The value of the `attribute_length` item indicates the attribute length, excluding the initial six bytes. (Therefore it must be 2+2**N*, where *N* is the value of `restrictions_count`.)
- The value of the `restrictions_count` item indicates the number of entries in the `restrictions` array. (There are further restrictions on this value, which are described below as restrictions on the length of the following `restrictions` array.
- The value of each `restrictions` item must be either zero (0) or else a valid index into the `constant_pool` table. Each item must correspond to a field type (for the sole item), the method return type (for the first item), or a method parameter type (for subsequent items). The `constant_pool` entry at that index (if not zero) must be a loadable constant *K*, which when resolved supplies type restriction information for a field value, method return value, or method parameter value for the field *F* or method *M* associated with this attribute. This constant *K may* be co-parametric with *F* or *M*.

A field or a method which only restricts its return type will only need one item in the `restrictions` array. A method which restricts one or more of its parameters will need more than one item XXX

There is a structural constraint on the length of the `restrictions` array. For a field, the array must not have more than one item. For a method, the `restrictions` array must not have more than 1+*N* items, where *N* is the arity of the method. (The arity counts `long` or `double` values once, not twice.) Type restrictions are applied only to types which correspond to non-zero items present in the array.

Thus, the array is allowed to be shorter than its maximum length, and is logically padded out with zeroes. Informally, the array must not be so long that it contains elements (whether zero or non-zero) that fail to correspond to restrictable types.

For a method, the first `restriction` item corresponds to the method return value, even if the method returns `void`.

Although type restrictions are envisioned as applying primarily to classes and interfaces, they may apply in the future to `void` or built-in primitive types. For this reason, the `restrictions` array includes entries which correspond to `void` returns and values of primitives like `int` and `long`.

It is recommended that the `restrictions` array be non-empty, and that it end with a non-zero item. However, the JVM must always be prepared to deal with either zero items, or non-zero items which resolve to ineffectual type restrictions (such as "any `Object`").

As will be seen, a type restriction on a field or method affects all accesses to that field or method.

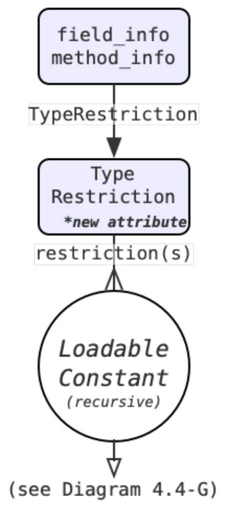
Any item in the `restrictions` array may refer to an invariant constant. If the field or method it applies to is parametric, any item in the array may also be co-parametric with that field or method. Parametric type restrictions are applied to parametric accesses and also to specialized instance fields or specialized virtual methods. Invariant type restrictions are applied to all accesses.

The effect of a type restriction is granular and independent for each field or method declaration. Type restrictions are not transferred to related API points, such as overriding methods in other class files. Type restrictions can affect the behavior of API points in a way that callers can see, since they can block callers from storing or passing or receiving excluded values.

There is no direct mechanism for acquiring the *specialized type* of a type-restricted field or method, although such data is surely useful. It is possible to envision special rules for `CONSTANT_MethodHandle` constants for the type-restricted API points that somehow use the `MethodHandle.type` property to encode type restrictions, but since type restrictions are not themselves types, this seems like the wrong tactic. For now the present, we will assume that, at the very least, there will be a reflective API to query type restrictions on API points. Perhaps a direct query can be created by building up a variation of the `CONSTANT_MethodType` item, which points to a type-restricted API point instead of a descriptor string.

The relations of the `TypeRestriction` attribute with other class file structures, including its reference to type restriction items in the constant pool, are summarized in Diagram 4.7-D(b).

Diagram 4.7-D(b). TypeRestriction attribute relations



(It seems possible, to some observers, that some version of the `TypeRestriction` attribute might appear in the future on a class or interface as a whole, rather than merely on its fields or methods. This might be the case if it would be useful to declare a parametric restriction that somehow applies to the class as a whole, rather than to its various members. Note that a species embodies such a whole-class type restriction; perhaps there are connections between species and type restrictions which are not yet fully understood.)

Specialized and/or parametric super types

A class's superclass and any implemented interfaces are collectively called *super types* (sometimes just *supers*).

Unlike fields and methods, whose types are declared via "flat" `CONSTANT_Utf8` descriptor strings, super types are indicated in the class-file by references into the constant pool, to `CONSTANT_Class` constants. This allows them to participate in specialization without requiring separate `Parametric` attributes.

Independently of whether a class or interface itself is parametric (i.e., has a `Parametric` attribute), any of its super types may be accompanied by a proposed linkage parameter, that is, by means of a `CONSTANT_SpecializationLinkage` constant which wraps a `CONSTANT_Class` constant.

As always, an interface may not specify any super class other than the mandatory `java.lang.Object`. In fact, it must also be a simple invariant `CONSTANT_Class`.

As always, the super class must be a symbolic reference which resolves to a class not an interface, and each super interface must be a symbolic reference which resolves to an interface not a class.

When a class or interface `C` has a specialized super `S`, the reference to `S` may take one of three forms:

- The super reference `S` is an invariant `CONSTANT_Class` constant, even though `S` has been declared as parametric. (For example, `OldList extends ArrayList`.) In this case, the class or interface is a subtype of the default "raw" species of the type named by `S`.

- The super reference `S` is both invariant *and* is a `CONSTANT_SpecializationLinkage` constant that wraps the name of `S`. (For example, `PointList extends ArrayList<Point>`.) In this case, the class or interface is a subtype of the species obtained by resolving `S`. The specialization information for `S` is recorded locally in the constant pool for `C` (specifically, in the resolution state of the `CONSTANT_SpecializationLinkage` constant for `S`). Before the class or interface `C` is loaded, the type `S` is loaded in its unspecialized form, temporarily ignoring the linkage constant. After `C` is loaded, when it is prepared, the species `S` is then resolved, using the linkage parameter resolved from `C`'s constant pool.

- The super reference `S` is parametric not invariant. In this case `C` must also be parametric, and `S` must be a `CONSTANT_SpecializationLinkage` constant co-parametric with `C`. (For example, `MyMaps<K,V> extends    ArrayList<Map<K,V>>`; note that the type variables don't have to "line up" exactly.) Before the class file for `C` is loaded, the type `S` is loaded in its unspecialized form, temporarily ignoring the linkage constant. Later on, when each distinct species of C is prepared, the reference S is specialized by the bootstrap method for the specialization anchor of C, and the JVM records the subtype relation between the species of S and C. (See the discussion of "s-tables" below.)

The third case, of a parametric super, amounts to a super-type restriction which is applied differently to different species.

As discussed elsewhere, the JVM immediately prepares a "raw" default species of any variant C that it loads. If this C has a co-parametric super S, then the JVM records the "raw" default species of S as the corresponding super for the default species of C, regardless of any structure of the constant pool reference for S. Thus, the co-parametric supers of a default species will be "raw all the way up".

This restriction simplifies the special processing of default species, especially in their role as "raw" universally compatible versions of specializable types.

Volume II: Linkage of Specializations (JVMS-5.4)

During execution, any given constant pool entry of type `CONSTANT_SpecializationAnchor` is prepared and (eventually) acquires a resolved value as a result of cooperation between two class files (containing two constant pools), the *caller* and the *callee*. The caller proposes a *linkage parameter value* for an API point in the callee. If the API point declaration is in fact variant over the anchor in the callee, the callee then *validates* the linkage parameter value, and selects or creates a specialization anchor object which embodies the specialization decisions resulting from the caller's request.

This cooperation occurs in the context of the resolution of the API point, which always includes a symbolic reference (to a class, interface, method, or field). The extra linkage parameter is resolved in the caller's constant pool, and then validated relative to the API point's declared anchor, as determined by the `Parametric` attribute of that API point in its declaring class file.

After resolution, the specialization anchor object is recorded as a permanent agreement between the caller and callee, specifically in the resolution state of the `CONSTANT_SpecializationLinkage` constant in the caller.

During execution of code in the callee, the specialization anchor object reappears as a value of the `CONSTANT_SpecializationAnchor` constant in the callee. The callee can make use of the specialization decisions embodied in the anchor by feeding the anchor as a proposed linkage parameter to its own `CONSTANT_SpecializationLinkage` constants, or by using `CONSTANT_Dynamic` constants to derive types, constants, and behaviors from the anchor.

Inheritance and overriding have additional effects on anchors, which are described elsewhere. It is always the case, however, that every symbolic resolution operation determines a particular API point declared in a particular class file, and *that* declaration controls the validation of any linkage parameter proposed by the caller.

Of course a class file can call one of its own API points, in which case the caller and callee would be a single class file. Nevertheless it is useful to clearly distinguish the responsibilities and actions of the caller from those of the callee. Sometimes we will use the caller-centric terms "local" and "remote" to describe the two perspectives, where the caller makes a "local" request to bind a anchor on a "remote" API point defined by the callee.

The validation process includes a decision (by the callee) whether to prepare a new resolution state for the `CONSTANT_SpecializationAnchor` constant, or whether to reuse a previously prepared resolution state. The JVM always supplies, as an option to the callee, a default specialization for every anchor, which is prepared at the same time as the invariant constants are prepared (during preparation of the class as a whole).

A "raw" symbolic reference, free of any involvement with `CONSTANT_SpecializationLinkage` wrappers, will always select such a default specialization, which in turn will operate (as far as the user can see) as if the compilation of generic classes still uses erasure as a translation strategy.

Validation always occurs relative to a particular anchor in a particular class file. In order to make the process of validation efficiently checkable and idempotent, the JVM defines a special type `SpecializationAnchor` (in package `java.lang.invoke`) which embodies validation of a linkage parameter, and all specialization decisions implied by that parameter. Each instance of this type is "locked" to a specific `CONSTANT_SpecializationAnchor` constant in a specific class file. As such, it is a pre-validated linkage parameter for any API point in that same class file that is parametric over the same anchor constant. It is invalid for all API points in other class files, or differently parametric API points in the same class file. However, the same `SpecializationAnchor` can be reused (efficiently, without revalidation) for multiple API points in the same class file, as long as they are co-parametric over a common anchor.

The internal structure of a `SpecializationAnchor` object can be organized so as to make frequent checking operations simple and fast. The important operations on it include finding the resolved values of derived parametric constant pool constants, checking that object instances have congruent species, perhaps finding "friend" specialization anchors (such as those for super- or sub-classes), and ensuring that `CONSTANT_SpecializationLinkage` states are correctly set up.

A `SpecializationAnchor` object is deemed validated "from inception". As soon as one is created, it is immediately valid with respect to the `CONSTANT_SpecializationAnchor` it is created with reference to. To protect encapsulation, all public factory methods for `SpecializationAnchor` require a full-power `Lookup` object for the class file that contains the `CONSTANT_SpecializationAnchor` in question. (The `Lookup` can be obtained from inside the class file, or by means of a privileged operation performed by a trusted language runtime.) This implies that returning a `SpecializationAnchor` object from a bootstrap method does not confer additional validity on it, but simply associates it with a particular client of an API point.

There are many potential language-specific aspects of the `SpecializationAnchor` object's API, such as a memoization of the originally proposed (yet unvalidated) linkage parameter value, or some sort of assembled metadata for use by reflection, or derived values such as species or specialized field and method types. It is clear that we cannot design in such aspects to the core API of `SpecializationAnchor`. It is thus an open issue (TBD) whether those aspects should be adjoined to the `SpecializationAnchor` API using inheritance or composition. In the case of composition, `SpecializationAnchor<T>` will be given a language-specific internal variable of type `T` which carries the weight of the language runtime's bookkeeping requirements, and instances are created (by the language runtime) with their partner object. In the case of inheritance, `SpecializationAnchor` is a more open class which can be subclassed by language runtimes, even though its constructor is (somehow) protected from arbitrary access. The case is complicated by the requirement that default specializations (representing "raw" types) should be created unilaterally by the JVM; this means that a specialization for the "raw" version of a class or interface must have a JVM-assigned class, not one determined during the course of a bootstrap call. It seems cleaner, for prototyping, to resort to composition, and give `SpecializationAnchor` a few one runtime-assigned variables. For JVM-created default specializations, the variables can be initialized to "boring" values like `null` (forcing the runtime to "just deal" with the annoying nulls) or else ask an anchor's bootstrap method to create the either all default specializations, or else the runtime helper objects for such specializations. But running bootstrap methods is not free; it can easily cause infinite bootstrap recursion if run in the early phases of class loading. For now, "just deal with the nulls" is the easiest way forward for JVM prototyping, but it seems a more equitable solution (allowing the runtime to participate in default specialization creation) is in the cards.

From the caller's point of view, a proposed linkage value can be any loadable constant pool constant. This loadable constant can be either known to be previously validated (e.g., a locally known `SpecializationAnchor` object) or some unvalidated value (which can be any object whatsoever). In the validated case, the constant value will always be a reference to an object of type `SpecializationAnchor`, produced by the anchor constant associated with the callee's API point.

In this design, validation is idempotent, not a transform from one type to another, from one point in an API scheme to another. It might seem cleaner to rigidly separate the "random junk" that callers propose for linkage parameters, from the validated `SpecializationAnchor` values, with separately typed API points for each. But such a design would satisfy no practical need, because in practice, every anchor proposal, at every API point, is tentative. This is true because specialization is an internal aspect of each API point, and can change (after recompilation of the API point declaration) at any time. Callers can guess at proper specializations, but the "handshake" between proposed parameters and validated anchors must be performed as a part of API point linkage. This is seemingly unfortunate, but the situation can be made much more tenable by ensuring that callers are *likely* to guess good (valid) anchors, and that the JVM can *quickly* revalidate them (without expensive bootstrap calls). In this setting, we don't need or want separate types; we expect that unvalidated values will (despite separate compilation and dynamic linking) quickly converge to validated values. And using separate types for both would only delay such a convergence.

Also, from the caller's point of view, a proposed linkage value can be either an invariant constant or a parametric constant. In the latter case, some previous caller must have already proposed a value for the variant constant's underlying specialization anchor, and that value was validated and agreed upon, so that there is a well-defined current resolution state for the variant constant.

Thus there are four cases for a caller's constant pool entry to propose a linkage parameter:

- validated, invariant: A `SpecializationAnchor` constant, which once determined is constant for all invocations of the caller. Example: A `SpecializationAnchor` object for the `CONSTANT_SpecializationAnchor` of a parametric interface `java.util.List`, denoting a species `List<Point>` for some other type `Point`.
- unvalidated, invariant: A value to be passed to an anchor's bootstrap method which is intended to request creation of some specialization, and/or a specialization anchor created for some other (perhaps related) API point. For example, a record instance requesting creation of `List<Point>` whose components are class mirrors for `List` and `Point`. Or, as another example, a `SpecializationAnchor` object for a class `java.util.List`, denoting (as before) a species `ArrayList<Point>`, which is being proposed as the linkage parameter for an API point (perhaps a constructor) of a subtype `java.util.ArrayList`.
- validated, variant: A constant of type `SpecializationAnchor` which is also parametric, and thus depends on an ambient anchor value (perhaps itself). For example, in the context of a generic method `Arrays.<E>sort`, the `CONSTANT_SpecializationAnchor` which determines `E`, and which is used to invoke some private, equivalently-parametric subroutine called (say) `Arrays.<E>mergeSortHelper`. Or, as another example, in the context of the same method, a call to a method in a helper class, `SortHelpers<T>`, where the linkage parameter is obtained from a constant (in the caller class `Arrays`) that reifies the type `SortHelpers<E>`, for each ambient value of `E` in `Arrays.<E>sort`.
- unvalidated, variant: A linkage parameter to be passed to an anchor's bootstrap method which is somehow dependent on some (previously determined) ambient specialization anchor. For example, in the context of a generic method `Arrays.<E>sort`, a `CONSTANT_Dynamic` constant whose input is the class mirror corresponding to the contextual value of `E`, and which computes the mirror of a derived type such as `E[]` or `List<E>`, to be further proposed as a type parameter value for some other API as part of the execution of `sort`.

Preparation of Constants

A constant is *prepared* when storage for its resolution state is assigned to it. When first prepared, most constants are in the unresolved state, but some are immediately set to some known value. A `CONSTANT_SpecializationAnchor` is always prepared resolved a `SpecializationAnchor` reference. (Constants which are not resolved, or which have trivial resolutions, may also be viewed as being prepared in a final state.) The rules for preparation of resolution states of constants are as follows:

- Invariant constants are prepared (if necessary) when their declaring class is prepared.
- Every `CONSTANT_SpecializationAnchor` is prepared once as a default specialization (in a "raw" empty state) when its declaring class is prepared.
- A `CONSTANT_SpecializationAnchor` is (subsequently) prepared in response to a library call (see `SpecializationAnchorBuilder` below) which creates a fresh specialization anchor for that specific anchor

constant. Such a call is typically the result of a validation request.

- Every constant *C* which is parametric over some anchor *R* is prepared exactly as many times as *R* itself is prepared. In fact, *C* and *R* is prepared at the same time as the associated `SpecializationAnchor` object is created. The resolution states of *R* and *C* can be accessed via that `SpecializationAnchor` object.

Thus, each `SpecializationAnchor` object serves as a handle on a set of consistently specialized constant resolution states. When a method executes bytecodes in the context of a caller-supplied `SpecializationAnchor` object, variant constants are determined relative to the resolution states of that same specialization anchor. As with all constant pool resolution states, these states start out in a neutral state, but eventually resolve to a permanent result, either successfully with a metadata reference or value, or else to a permanently recorded resolution error.

The information content of a `SpecializationAnchor` object includes the following items:

- **Anchor identity:** An internal reference to the metadata describing the `CONSTANT_SpecializationAnchor` constant *R* which it binds, within the run-time constant pool of the particular loaded `class` file *F* that defines it.

- **Parameters:** One or more arbitrary object references permanently associated with the anchor when this specialization created by the language runtime. The JVM assigns no particular meaning to the runtime value. It may be a list or tuple of type mirrors, for example. In the special case where this object represents a JVM-created default specialization, only a `null` reference is visible.

- **Species:** A species object which represents this specialized class in which this specialization is situated. If the class is not specialized (with respect to this anchor), then the “raw” `Class` mirror is reported instead. (Or null? TBD.)

- **Dependent constants:** A set of resolution states, one for each constant *C* which is parametric over *R*. (Constants bi-variant *R* and *Q* are omitted from these resolution states if *R* is the outer anchor to *Q*.) These states are prepared and added to the run-time constant pool of the loaded `class` file *F* when the `SpecializationAnchor` object is created.

- **Outer specialization:** If the anchor *R* is not of kind `PARAM_MethodAndClass`, a `null` reference. Otherwise, a second `SpecializationAnchor` object which specializes *R*’s outer anchor *Q*, which is of kind `PARAM_Class`. Note that the constant pool states for this outer specialization may be shared by many specializations of *R*.

- **Associated class:** A reference (of type `Class<?>`) to the particular class declared by the `class` file *F*, and containing the specialization anchor *R*. (This value is logically derived from the anchor identity, but may be physically present in a field of the `SpecializationAnchor` as an implementation artifact.)

- **Associated fields:** A set of methods which are parametric over the anchor of this specialization, along with their type restrictions. (This is for reflection only, and may be safely omitted while prototyping.)

- **Associated methods:** A set of methods which are parametric over the anchor of this specialization, along with their type restrictions. (This is for reflection only, and may be safely omitted while prototyping.)

- **Associated default:** The unique `SpecializationAnchor` object representing the default specialization for the anchor *R*. A default specialization points to itself as its associated default. (This value is logically derived from the anchor identity, but may be physically present in a field of the `SpecializationAnchor` as an implementation artifact.)

The data structure itself appears to require about five fields per distinct `SpecializationAnchor`, plus an array element for each distinct resolution state of the dependent constants. It seems likely that preparation of resolution states can handled with a simple Java object array allocation of an appropriate size, with suitable conventions for distinguishing unresolved, resolved, and erroneous states.

Resolution of `CONSTANT_SpecializationLinkage` constants

Any use of a `CONSTANT_SpecializationLinkage` in place of the symbolic reference that it wraps first resolves the symbolic reference to an API point *M*.

Next, if the API point *M* is not parametric, the result is as if the `CONSTANT_SpecializationLinkage` constant were not present, but rather the “raw” symbolic reference had been used from the start.

If the API point *M* is parametric over some *R*, then the proposed linkage parameter value referred to by the `CONSTANT_SpecializationLinkage` constant is resolved. The resulting value is then validated against *M*’s anchor *R*, using a bootstrap method (declared on *R*) if necessary.

After successful resolution (including validation) of the `CONSTANT_SpecializationLinkage` constant, both resolved components are permanently recorded by the JVM: the symbolic reference, and the specialization anchor object. In the case of unsuccessful resolution, the appropriate `Error` object is recorded for future uses of the `CONSTANT_SpecializationLinkage` constant.

For example, if a `CONSTANT_Class` constant for some *C* is wrapped in a `CONSTANT_SpecializationLinkage` constant, and the latter is resolved, then first *C* is resolved, and then if *C* is parametric (which is likely), the linkage parameter value proposed by the `CONSTANT_SpecializationLinkage` constant is immediately validated against *C*’s anchor. The resulting specialization anchor object is then permanently recorded with the `CONSTANT_SpecializationLinkage` constant. The net result is that a specialization of *C* has been determined in the client’s constant pool. The resolved constant may be used with various bytecodes, such as `ldc` (to load the species of *C*), `new` (to make a specialized instance), `instanceof` (to test an object whether it conforms to that species of *C*), or as part of a symbolic reference to one of *C*’s members.

Thus, the resolution state of a `CONSTANT_SpecializationLinkage` constant records not only the identity of a remote API point, but also the specialization decisions appropriate to that remote API point.

The full resolution state of a `CONSTANT_SpecializationLinkage` constant is never accessible as a loadable constant (`CONSTANT_Dynamic` argument or `ldc` bytecode); it is opaque to the caller except in the type restrictions of the parametric API points, and (of course) in their behaviors.

In one case only, a `CONSTANT_SpecializationLinkage` constant can serve as a loadable constant, and that is when the constant it wraps is already a loadable constant, that is, a `CONSTANT_Class`. In that case, the value from the linkage constant is the species derived from the specialization anchor (as if by `SpecializationAnchor.species`). Note that a single species may, in some cases, be associated with several class specializations.

A previous version of this proposal exposed an anchor object as the constant value of a `CONSTANT_SpecializationLinkage` constant. This behavior would be contrary to the goal of encapsulating specialization decisions. The class that produces a `SpecializationAnchor` object may choose to expose it through a public static API point. Such decisions, made by whatever runtime system implements the bootstrap method, are outside of the JVM's purview.

Validation of Linkage Parameter Values

As the latter part of resolving a `CONSTANT_SpecializationLinkage` constant, its proposed linkage parameter value is resolved and validated, against the remote parametric API point resolved from the symbolic reference.

If the remote API point is not parametric, the linkage parameter is neither resolved nor validated (because there is no anchor constant to validate it); instead it is ignored. (This is not an erroneous state; an API point is always free to ignore proposed linkage parameters.) As a loadable constant, the resolved value of such a `CONSTANT_SpecializationLinkage` constant is a placeholder value supplied by the runtime (TBD, probably `null`) which indicates that the resolved API point was, in fact, invariant.

Otherwise, the API point is parametric over an associated `CONSTANT_SpecializationAnchor` defined in its class file. In that case, a proposed linkage parameter is defined as valid for that API point if and only if it is a reference to a `SpecializationAnchor` object that was created for that anchor, either by the JVM (as the unique default specialization for that anchor) or by successful invocation of the anchor's bootstrap method.

For a remote API point parametric over some anchor *R*, if the proposed linkage value *V* is valid for *R*, then the resolution state of the `CONSTANT_SpecializationLinkage` constant records *V*. Such a *V* may be called "pre-validated". The simplest example of pre-validation occurs is when the `CONSTANT_SpecializationLinkage` constant proposes a `CONSTANT_SpecializationAnchor` local to the class file. (This is not automatic: If a translation strategy fails to "thread through" a local anchor value to another local API point usage, then the "raw" default specialization is selected for that API point.) It may also happen if a pre-validated `SpecializationAnchor` is obtained from some other source (via condy), and placed in the constant pool where a `CONSTANT_SpecializationLinkage` constant can propose it.

If the API point is class-variant, then a `SpecializationAnchor` for a bi-variant anchor (in the same class file) is treated as pre-validated, as well as a `SpecializationAnchor` for the class anchor itself.

As a special case, if *V* is the null reference, and the remote API point is parametric over some anchor *R*, then the JVM substitutes the (internally known) default `SpecializationAnchor` reference for that value, and records the latter reference as the pre-validated value.

In all other cases, the proposed linkage parameter will be something like a quasi-symbolic package of type mirrors, which the anchor's bootstrap must validate and map to a species or other specialization information.

Suppose the remote API point is parametric over some anchor *R*, but the proposed linkage value *V* is not validated for *R*. In that case, the bootstrap method for *R* is invoked. The bootstrap method receives the proposed value *V*. It is expected to return a `SpecializationAnchor` object reference valid for *R*, else a linkage error will be raised. The possible non-erroneous outcomes are:

- A freshly created `SpecializationAnchor` object (over *R*) is returned. The JVM notes the fresh creation, and prepares fresh new constant pool states for every constant pool entry parametric over *R*. (In this case, any parametric constants of *R* will be re-resolved, if and when the API point referred to by the `CONSTANT_SpecializationLinkage` constant makes use of them.)
- A reference to a default `SpecializationAnchor` object, created for *R* when its `class` file was prepared, is returned. (In this case, parametric constants of *R* will continue to be resolved according to the prepared resolution states of that special object.)
- A reference to some other pre-existing `SpecializationAnchor` object (over *R*) is returned. (In this case, parametric constants of *R* will continue to be resolved according to the prepared resolution states of that pre-existing object.)

Note that in all these non-erroneous cases, the returned reference is to a `SpecializationAnchor` object which in fact would be pre-validated in a subsequent linkage request to the same API point.

The erroneous cases are as follows:

- The bootstrap method returns something bad: A `SpecializationAnchor` object for some anchor other than *R*, or a null reference, or some object which is not a `SpecializationAnchor`. In this case, a `BootstrapMethodError` is raised instead, just as if the bootstrap method had thrown that error. (See next case.)
- The bootstrap method throws an exception *E*. In this case, validation has failed, and the resolution of the `CONSTANT_SpecializationLinkage` also fails with the exception *E* (if *E* is an `Error`) or else a `BootstrapMethodError` wrapping *E*. (No access to the API point is possible via this `CONSTANT_SpecializationLinkage` constant, and

so there are no resolution states or dependent constants to worry about.)

The null reference is not allowed as a return from the BSM even though it is allowed as a pre-validated sentinel selecting a default specialization. Also, a bi-variant `SpecializationAnchor` is not allowed as a return from a BSM which is acting for a `PARAM_Class` anchor. The return value must be a `SpecializationAnchor` exactly for the requested anchor constant.

If an API point is parametric but it is used via a "raw" symbolic reference (not augmented by a `CONSTANT_SpecializationLinkage` constant), the the anchor is linked to its default "raw" instance, just as if a `CONSTANT_SpecializationLinkage` constant *were* used, but had proposed the default specialization for *R* (or `null`, equivalently). According to the rules above, this effective proposed value is in fact pre-validated, and does not require a bootstrap method call.

An earlier version of this proposal called for the bootstrap method to be executed, but on further consideration this seems to be a feature which is both error-prone and not particularly useful. Instead, the JVM guarantees that all API points always accept invariant uses, applying the JVM-supplied default `SpecializationAnchor` to them. One might wish for a way to prevent some API points from accepting "raw" default specializations, but truly effective prevention of raw access appears to be a research project, rather than something achievable by a simple design decision. (Part of the research would be to decide how to adjust the core reflection API, such as `jlr.Method.invoke`, which currently requires some sort of default specializiaton setting for API points that it reflects.) Note that translation strategies which aim to avoid default specializations (for selected API points) can partially avoid default specializations by compiling default-rejecting guards into their methods and other paths, and can spell names in such a way that legacy code cannot accidentally link to them.

The default specialization can also be used (if desired) for the behavior of all kinds of malformed API accesses, allowing out-of-date or "type polluted" clients to quietly fall back to the raw behavior of the desired API point. This behavior is fully under the control of the bootstrap method. The only behaviors "hardwired" into the VM are where a non-parametric API point silently ignores a proposed linkage parameter, or where a pre-validated `SpecializationAnchor` (such as the "raw" default or a previous BSM result) is proposed during the resolution of a `CONSTANT_SpecializationLinkage` constant.

There is some expressive value in allowing the symbolic reference of a `CONSTANT_SpecializationLinkage` constant to have its own variance, so that a method of a parametric class can be additionally parametric. To allow this, the JVM accepts a `CONSTANT_Fieldref`, `CONSTANT_Methodref`, or `CONSTANT_InterfaceMethodref` constant to refer to a `CONSTANT_SpecializationLinkage` constant wrapping a `CONSTANT_Class`, where it would normally simply point to a "raw" `CONSTANT_Class` symbolic reference. Thus, references to fields and methods (but not plain types) can inject specializations into their class scopes (by wrapping the embedded `CONSTANT_Class`) or directly into the field or method (by wrapping the constant as a whole). In fact, linkage parameters can be proposed *at both positions*.

There is a loose relation between code and layout customization and constant preparation. If the validation of a linkage parameter results in a fresh `SpecializationAnchor` object with freshly prepared resolution states for parametric constants, then the VM has the *option* to internally customize code and/or data layouts to those states. VM implementations can refrain from exercising such options. Conversely, if validation results in the use of a previously created `SpecializationAnchor` object (such as the "raw" default specialization for that anchor, or perhaps some general-purpose species for a group of erased types), then the VM *must* use the shared (perhaps unspecialized) states of the parametric constants in that `SpecializationAnchor` object.

There may be library routines and/or optimization directives, which affect the VM's decisions to specialized code and/or layout to particular instances of `SpecializationAnchor`. In some cases, the parametric constants within a `SpecializationAnchor` will not require any associated code or data specialization; in those cases, the shared code and data will simply make use of the parametric constant values as if they were extra invisible arguments to methods and extra invisible (final) fields in data.

Within a stack frame executing a parametric method, that method's `CONSTANT_SpecializationAnchor` constant resolves (for that execution only) to the `SpecializationAnchor` reference validated by the caller when the caller symbolically resolved its API point reference to the method.

Additionally, within a stack frame executing a parametric method within a parametric class or interface, that class or interface's `CONSTANT_SpecializationAnchor` constant resolves (for that execution only) to the `SpecializationAnchor` reference validated by the caller when the caller symbolically resolved its API point reference to the method.

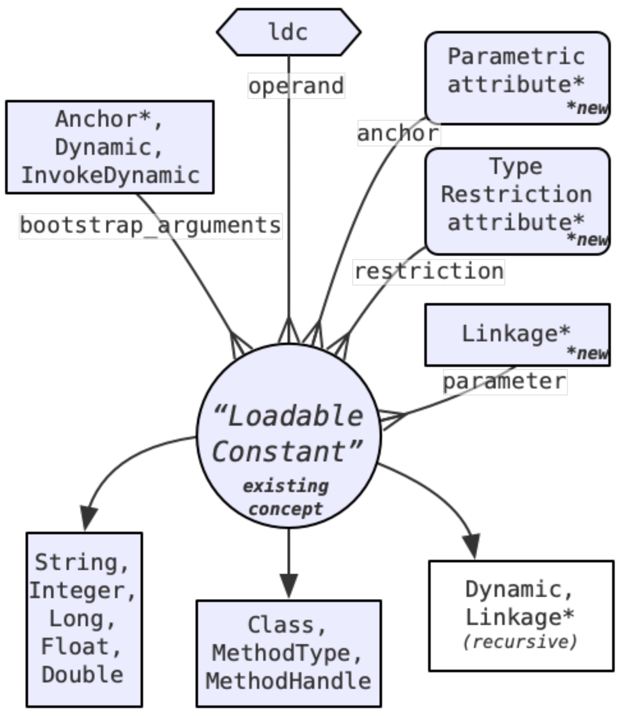
Thus, an `ldc` of a `CONSTANT_SpecializationAnchor` produces a low-level `SpecializationAnchor` object. At the option of the translation strategy, the reflective API of this object can be used to obtain additional relevant information, such as species or type variable bindings.

For example, the `value` method of `SpecializationAnchor` returns a value assigned by the bootstrap method that created the `SpecializationAnchor` (or `null` for the JVM-created "raw" default `SpecializationAnchor`).

Non-parametric methods do not have access to parametric constants, not even if the non-parametric method is declared in a parametric class. In order for a `CONSTANT_SpecializationAnchor` constant to resolve in a method, that method must be parametric over that anchor. Specifically, in order for a `CONSTANT_SpecializationAnchor` constant of kind `PARAM_Class` to resolve in a method, that must be either co-parametric with its enclosing class, or or bi-variant. In the latter case, it can resolve the values of either or both of the relevant `CONSTANT_SpecializationAnchor` constants (one for the class, and one of kind `PARAM_MethodAndClass`).

The set of loadable constant pool constants (both old and new) is summarized in Diagram 4.4-G. These constants are usable as bootstrap method arguments, with the `ldc` family of instructions, and as specializations proposed via `CONSTANT_SpecializationLinkage`.

Diagram 4.4-G. “Loadable Constant” relations


ldc
operand
Parametric attribute* *new
anchor
Anchor*, Dynamic, InvokeDynamic
bootstrap_arguments
Type Restriction attribute* *new
restriction
Linkage* *new
parameter
“Loadable Constant” existing concept
String, Integer, Long, Float, Double
Class, MethodType, MethodHandle
Dynamic, Linkage* (recursive)

Note: A Linkage constant is only loadable if its reference is loadable, i.e., a Class. Other Linkage constants (e.g., of a Fieldref) are not loadable.

Bootstrap upcall details

Bootstrap method calls occur when specialization anchors are required, but non-valid linkage parameters values are proposed. This usually occurs in the context of dynamic linkage between a caller and a callee, where the proposed linkage parameter is not already valid for the callee.

Bootstrap methods are made in the usual way, as if by `MethodHandle.invokeWithArguments` on the specified bootstrap method, with a leading full-privilege `Lookup` argument followed by fixed task-specific arguments, and any static arguments following last.

It is the responsibility of the language runtime, not the JVM, to ensure that the results returned by bootstrap method calls are valid for the context of the JVM event which caused the bootstrap method call.

Default specializations are created for each anchor in a class’s constant pool during preparation of that anchor. For a `PARAM_Class` anchor, this occurs before execution of any class initializer. These default `SpecializationAnchor` instances are created automatically by the JVM, and their characteristics are set automatically, without any appeal to any bootstrap method.

For any `SpecializationAnchor` object (default or not) for a `PARAM_Class` anchor *Q*, and for any anchor *R* of kind `PARAM_MethodAndClass` in the same class-file, the JVM automatically creates (as required) a default specialization for *R*, without the intervention of a bootstrap method, which represents the “raw” default version of *R* in the context of *Q*’s specialization (default or not) represented by the first `SpecializationAnchor` object. Such an “inner” default specialization for *R* is needed as a bootstrap method argument whenever a *R*-variant method is to be specialized.

All other `SpecializationAnchor` objects are created by runtime code, and injected into constant pools as the result of bootstrap method invocations.

Validation bootstrap calls
When some given API point is accessed, validation bootstrap calls occur when all of the following conditions are true:

- The symbolic reference of the API point *M* resolves to a declaration located in the `class` file of some class or interface *C*.
- Access checking of *M* succeeds (relative to the caller).
- The declaration of *M* is parametric over some *R*.
- A proposed linkage parameter value *V* is present (via a `CONSTANT_SpecializationLinkage` wrapper on the symbolic reference for *M*.).
- *V* is not already a `SpecializationAnchor` validated for *R*.
- Even if *V* is already a `SpecializationAnchor` validated for some related *Q*, a valid `SpecializationAnchor` for *R* cannot be derived from *V* automatically. (There is one case of such automatic derivation described below.)

When these conditions all prove true, we may say that “*M* requires a validation bootstrap for *V*”. In such a case, the bootstrap method for *R* is invoked on these arguments:

1. A full-privilege `Lookup` argument for the class or interface *C*.
2. The default `SpecializationAnchor` object *B0* for this anchor *R*, as previously generated internally by the JVM. (If *R* is bi-variant, its outer link may or may not be default, and in any event carries the result of a previous bootstrap.)
3. The proposed linkage parameter value *V*, which has failed to validate to *R*.
4. …Any static arguments associated with the bootstrap method.

Note that the specific identity of *M* is irrelevant to the validation bootstrap. It may be the case that several methods in *C* share a common anchor; any one of them could trigger the same validation bootstrap, and all of them could contrive to use the single `SpecializationAnchor` result in common, if they call each other using their `CONSTANT_SpecializationAnchor` constant.

In order for the validation to succeed, the following conditions must all hold true:

- The bootstrap method call returns normally.
- The result of the call is a reference to a `SpecializationAnchor` object *B*.
- The JVM can observe that *B* was created for *R* (as defined by *C*).

The “raw” default specialization *B0* is always a legitimate return value for this bootstrap method call. If returned, that specialization *B0* selects, on behalf of the requesting client, the default unspecialized behavior that the JVM would assign to the API point if the client had not proposed any linkage parameter value *V*.

Otherwise, the validation will fail with an instance of `Error`. A `BootstrapMethodError` will be created, if no instance of `Error` is already being thrown.

The bootstrap when *M* is a field is provided mainly for symmetry with the other cases. For parametric field references, it is expected that the linkage parameter *R* will be injected into the `CONSTANT_Class` component of the `CONSTANT_Fieldref`, and not at “top level” on the `CONSTANT_Fieldref` itself. If this feature proves incrementally difficult to implement, it can be omitted.

The bootstrap method may consult the API of *B0* to learn various details about the structure of the anchor being specialized and its class file. This uses a Reflective API available on all `SpecializationAnchor` objects.

The bootstrap method may use a factory API for `SpecializationAnchor` objects to create a brand new specialization. This new specialization by default will possess a new species, if *B0* is class-variant. The factory API also allows the bootstrap method to link two specializations together sharing a common species.

The bootstrap method is responsible for validating the proposed linkage parameter *V*, and for storing appropriate parameter information in standard locations on the resulting `SpecializationAnchor` object. In this way, even if the specialization anchor is lost, a species all by itself can serve as a “key” to recover the same specialization state, or an equivalent one.

The JVM may supply a fast path for validating a species when presented as a linkage parameter, expanding it into a corresponding specialization anchor.

Reflective API of `SpecializationAnchor`
Principally for the use of bootstrap methods, a substantial amount of information is exposed by the API of the Java type `SpecializationAnchor`. For any given specialization *B*, for an anchor *R* in the class-file of a class or interface *C*, the following data are defined and exposed by the JVM through *B*’s API:

- The class *C* which declared the anchor for this anchor.
- An opaque numeric value which uniquely identifies *R* (within *C*).
- Whether *R* is of kind `PARAM_Class`, `PARAM_MethodAndClass`, or `PARAM_MethodOnly`.
- A *parametric super list* describing all parametric supers of *C*. (This list will be empty unless *R* is of kind `PARAM_Class`.)
- A *parametric field list* describing all fields *F* which are co-parametric with *R*. (This list will be empty unless *R* is of kind `PARAM_Class`.)
- A *parametric method list* describing all methods *M* which are co-parametric with *R*.
- If *R* is of kind `PARAM_MethodAndClass` parametric over a *Q* of kind `PARAM_Class`, a `SpecializationAnchor` object for *Q* that supplies the class context for *B*.
- The corresponding “raw” default specialization *B0* for *B*.

In fact, the default specialization *B0*, in common with all (present and future) specializations of *R*, supports reflective queries which expose all API points in *C* (including but not limited to *M*) which are parametric over *R*. Their names and descriptors are available to the bootstrap method logic. Also, the JVM exposes, via *B0*, the resolved values corresponding to the `anchor_index` in the `Parametric` attribute of each API point. (If there is no such constant, the corresponding value is reported as a null reference.)

For additional concrete details, see the section Sample bootstrap API below.

Because none of the above information is changed by specialization, all `SpecializationAnchor` objects *B0*, *B*, etc., for a given anchor *R*, whether default or not, report the same reflective information about *R*.

A specialization *B* can be tested whether it is a JVM-prepared default specialization *B0* simply by testing whether a query the corresponding “raw” default yields *B* itself again.

Thus the default specialization *B0* can also serve as a reflective proxy unambiguously identifying the anchor *R*. This may be useful if the bootstrap wishes to perform some kind of reflection on the `class` file to gather more information about the anchor, such as which parts of class refer to it via `Parametric` attributes. Such additional reflective queries are TBD, and are not necessary for Java generics. The API for `SpecializationAnchor` may also (TBD) include (privileged) queries about which constant pool structure, in which `class` file, it corresponds to.

As described below, if *M* is bi-variant, *B0* may be a regular default specialization for *R* (which is of kind `PARAM_MethodAndClass`), or an “inner default” which is previously specialized to some non-default outer specialization (of kind `PARAM_Class`). Thus, *B0* can carry “outer” information from an enclosing parametric class, allowing the bootstrap method to consult the details of the enclosing specialization in case they are relevant to the further “inner” specialization of *M* (and its co-parametric siblings under *R*).

The bootstrap method is allowed (though not required) to use all relevant reflected data to create a new specialization encoded in a fresh `SpecializationAnchor` *B* which assigns arbitrarily specialized types and values to each of the API points parametric over *R*. The specialized types are enforced on all clients which use *M* (or any of its co-parametric siblings) via such a *B*. The specialized values are freely available (as `ldc` constant values) to all clients which link to the API points via *B*.

The parametric super, field, and method lists are all simple arrays. The elements of these arrays are presented in an arbitrary order selected by the JVM.

The parametric super array contains nested array items of the form `{x,s}`, where each `x` is a `Class` mirror for a super *S* of *C* that was declared (using a `CONSTANT_SpecializationLinkage` wrapper) as co-parametric with *C*, and each `s` is a *specializer datum* for *S*. The specializer datum is derived from the constant pool structure of the co-parametric reference *S* in a form which a bootstrap method can inspect and execute as needed. (The design is TBD; it may be a reflected `ConstantDesc` for the variant constant `S`, or perhaps a functional transform object.)

Factory API for `SpecializationAnchor`
When a new `SpecializationAnchor` must be created, the bootstrap method is responsible for marshalling all specialization decisions and handing them to a factory method, which then creates a fresh `SpecializationAnchor` object which can then serve as a record of those decisions, and a location where specialized constants can be derived.

The factory method takes the following arguments:

- A full-power `Lookup` object on the class declaring the anchor, enabling the privilege of creating a new specialization.
- A default `SpecializationAnchor` for the same anchor constant, serving as a template for the new specialization.

For an example, see `SpecializationAnchorBuilder::start` below.

The factory method returns a builder object which holds a “larval” `SpecializationAnchor` object of the correct shape.

While the specialization object is larval, the builder object can be requested to initialize the specialization object's record of parameter bindings and species. (Other actions are TBD.)

This information is stored permanently in a newly created `SpecializationAnchor` object and returned to the bootstrap method.

When the builder object is told to finish building, it returns the same `SpecializationAnchor` object, now permanently in a state usable by the JVM.

If the anchor is of kind `PARAM_Class`, the JVM also creates a species object (of type `Species` or perhaps `Class`, TBD) which embodies and reflects the decisions about supers and field types. The `species` method of `SpecializationAnchor` provides access to this JVM-created value.

The loadable constant value of a `CONSTANT_SpecializationLinkage` constant in any client which links to this species is that species. This is the only case of a `CONSTANT_SpecializationLinkage` constant functioning as a loadable constant.

The species is automatically created by the JVM when the builder finishes the object, if none was previously requested via the builder API.

The bootstrap method is free at any time to discard the builder and the larval anchor object, and return some other (compatible) anchor object to use instead.

Effects of type restrictions on parametric fields and methods
The JVM carefully records the association of type restrictions with specialized fields and methods. It enforces field type restrictions by requiring all stored values (even the initial default value) to conform to the restriction. It enforces method argument type restrictions by casting (or otherwise checking) all passed arguments before method entry (and even before virtual method selection). It enforces method return type restrictions by casting (or otherwise checking) all returned values on method return.

When a field is written, or a method parameter is bound to a value, its type restriction (if any) is applied as a runtime check. Again, when a field is read, or a method return value is received, a type restriction is applied as a runtime check. In any case, when such a runtime check fails, the access is aborted and a subclass of `RuntimeException` is thrown.

For example, a failed check may be reported via a `ClassCastException` or `NullPointerException` or `IllegalArgumentException`, depending on the nature of the type restriction.

Furthermore, even in the case of unspecialized ("raw") access, "raw" values stored to specialized instance fields and "raw" specializations passed to specialized methods are subject to type restrictions derived from the specialization of the containing object, as determined dynamically. Untyped reflective APIs also enforce type restrictions.

In the case of method overrides, two sets of type restrictions are applied, the type restrictions (if any) for the resolved symbolic reference to the method (at its call site) and also the type restrictions (if any) for the selected method. Type restrictions on unresolved, unselected methods are ignored by virtual calls.

In keeping with the order of operations in a virtual method call, type restrictions on the resolved methods are applied to arguments before type restrictions of the selected method. Similarly, a type on the return type of the selected method is applied before a type restriction on the return type of the resolved method.

The JVM may simplify the checking process if it can determine that the type restrictions on the resolved and selected methods are somehow identical or compatible. It does not enforce any kind of compatibility on resolved and selected methods.

For non-virtual calls (special and static) only the type restrictions of the resolved method are consulted.

The JVM is allowed but not required to use type restrictions to customize internal implementation choices about field layout and method calling sequence. Whether or not it does so, it must enforce all type restrictions, whether invariant or parametric.

The effect of such type restrictions is to allow (though not require) the JVM to organize the storage and representation of fields, arguments, and return values to "fit exactly" into the restricted types. Primitive values can be unboxed and stored directly in object layouts or registers.

In at least some circumstances (discussed below), a restricted type is allowed to be disjoint from the declared type of the field or method. In such a case, the field or method is inaccessible (in that particular specialization). Such a field need not occupy any space in an instance layout, and such a method cannot be invoked (or cannot return) without an exception.

Because default specializations contain no type restrictions, the only field and method types that matter are those reported by the descriptor strings of the `field_info` and `method_info` structures. Again, because of this, legacy clients of parametric classes will always see the unrestricted versions of their various API point types. This lack of type restrictions is one reason we informally refer to default specializations as "raw".

Current JVM implementations usually contrive to the layout of objects so that each (non-static) field has a unique offset within all objects that contain that field. Doing this requires (typically) a prefixing scheme where the fields of each superclass precede the fields of any of its subclasses, in the order of memory layout within instances of such a subclass. Parametric field types disturb this tidy algorithm, since a superclass can introduce a field whose size varies from species to species, thus perturbing the otherwise-constant field offsets in all the subclasses. This tidy algorithm can be rescued by a simple expedient: Always allocate all parametric fields (or at least, all size-variant fields) after all invariant fields. The ordering in the instance layout would thus be all invariant fields in super- to sub-class order, followed by all variant fields, in some arbitrary order (convenient to the JVM). Locating a variant requires an extra indirection somewhere to find a field offset, and accessing it will in general require another indirection to determine its type and/or size.

Type restrictions are enforced by the JVM *in addition* to each corresponding type enforced by the verifier. This enforcement must amount to a pointwise narrowing, as if by `checkcast`, of each field, argument, and return type. (Any enforcement which cannot be simulated by `checkcast` is not attempted, but rather leads to an error.) This design preserves stack effects and types mandated by the verifier.

Because the API type information is co-parametric with the API point, and is enforced exactly wherever this API point is, the JVM is allowed to construct customized calling sequences or layouts for specialized API points, if it so chooses.

If we were to allow a method returning a non-`void` *T* to specialize to a `void`-returning method, the preservation of verifier effects would require that a default value of *T* be pushed on the stack. It seems simpler to disallow any change which would change stack effects, such as changing between non-`void` and `void`, or changing the arity of a method type, or changing between a primitive type and any other type. Although it does not seem to buy us anything, we could also reject valid type conversions that are widenings (e.g., from `Number` to `Object`) or are not proper narrowings (e.g., from `Number` to `Comparable`). We can revisit these questions as we further converge primitives with class types.

The second set of enforcements, to specialized types, is performed dynamically, at each invocation or access, by referring to information stored in the link resolution state at each specialized use point.

The effect is analogous to that of statically inserted casts, in previous versions of Java generics. Unlike those previous versions, the casting depends not on a static decision by the bytecode compiler, but rather by a request from a caller who wishes for a particular specialization of a parametric API point, after runtime linking to the declaring class of that API point. There are no bytecoded casts at specialization points, and generic code is capable of making specialized type restrictions as well as client code. Also, the final decisions about type restrictions are made by the generic API point declaration, and not by its callers.

It may be useful that if a specialized type constant resolves to the type reflector `void.class` (or some other sentinel value), the corresponding API point would produce a `LinkageError` when used with the same specialization. This unpassible type restriction would make the API point inpossible to use for that specialization.

This provides a useful way to translate "logically optional but physically required" methods and fields. For example, if either a field or method happens to specialize to `void`, it becomes unlinkable (in that specialization). The exact encoding of a vacuous type is TBD; it may use `void.class`, `null`, or some other special token.

If the narrowing of types proves to be inconsistent in some other way, an error (such as a `LinkageError`) will also be reported. (In this case an appropriate `BootstrapMethodError` might be a useful diagnostic. This is TBD.)

Reflective APIs will provide access to specialized types assigned to specialized API points. The special case of `void` is likely to map to a sentinel value (such as `null`) meaning "no valid type is available".

Note that methods which must return a `null` value could be encoded using a hypothetical `NullReference` token in return position, if that is an important use case. Likewise, specializing a field to this token would amount to deleting it from the specialized layout, and forcing `getfield` to return a constant `null`. This may be useful for solving some compatibility problems, where a rarely used legacy field must still be accessible to `getfield`.

Unpassable restrictions and impossible values
Because type descriptors are static while type restrictions are dynamic, it is possible that a type restriction on a field or method can conflict with the type descriptor on a field or method, to the extent that no value that is compatible with the type descriptor is also compatible with the type restriction. The JVM does not treat an incompatible type restriction as a malformed input, but simply enforces it as fully *unpassable*, declaring its corresponding values to be impossible. An impossible method parameter is not simply dropped from a calling sequence; it prevents any call to the method from ever getting started. Simiarly, an impossible return value is not simply omitted from a method's result; it prevents any call to the method from returning normally. An impossible instance field is not simply omitted from an object layout; it prevents the object as a whole from ever being instantiated. (Alternatively, such an impossible field can be treated similarly to a failed resolution of a type restriction, which would fail a class loading operation in the case of an invariant restriction, or a specialization operation in the case of a parametric restriction; this is TBD.)

Other upcalls
Although specialization creation (of non-default specializations) is the main focus of bootstrap calls, there are other kinds of upcalls which are performed in the usual course of executing parametric code. Here is a list; further details are presented elsewhere in context:

- *Constant derivation:* A parametric `CONSTANT_Dynamic` constant can be used to compute and cache useful values which are dependent on a validated linkage parameter. The bootstrap method for such a constant may refer directly to a `SpecializationAnchor` object, or (less directly) to a type species, or a specialized field or method type, or a derived type or species (such as `List<T>` or `T[]` from `T` or vice versa). All of these operations can be assembled from appropriate bootstrap methods, plus calls to the `SpecializationAnchor` API.

- *Virtual dispatch:* When a virtual call selects a method other than its statically resolved method (i.e., an override), *and* that overriding method is parametric, linkage parameter revalidation must be performed. In this case, the JVM gives the language runtime wide latitude for invoking the overridden method. It performs an upcall to the `SpecializationAnchor` object which is statically present on the call, and permanently records the result of the upcall as a virtual call connector, in association with the constant pool structure which was used to make the virtual call. All future virtual calls (to parametric overrides, from that particular call site) are handled by this connector, by means of upcalls that originate from the JVM but are implemented in the code of the virtual call connector.

- *Virtual fields:* A field reference instruction might access (read or update) a field, but the containing instance specialization might not be identical with the specialization required by the field. This can occur, for example, if the field reference is specialized to particular container species, but the container itself is the default species for the container class. On the other hand, the field reference might be "raw" (thus validating to the default specialization of the class) yet the instance is specialized to some non-default specialization. In all such cases, the JVM could give the language runtime wide latitude for performing the field access. It might perform an upcall to the `SpecializationAnchor` object which is statically present on the field reference, and permanently records the result of the upcall as a virtual field connector, in association with the constant pool structure which was used to make the field access. All future field accesses (that do not exactly match the intended anchor) are handled by this connector, by means of upcalls that originate from the JVM but are implemented in the code of the virtual field connector.

- *Type testing:* A parametric `instanceof` instruction (or any equivalent type test) could be implemented by an upcall on the species object's `isAssignableFrom` method, except in cases of exact match. This would allow the language runtime to implement appropriate subtyping rules, such as those which make the default specialization for a `KIND_Class` anchor appear to be a supertype of all other specializations. It may (for some languages) also allow species to have more complicated subtyping rules. For now, it seems best to build a couple of really obvious rules, directy inside the JVM, and see how far that takes us.

- *Array creation:* The JVM is capable of creating an array for any plain class or interface, without special assistance. Its abilities to create arrays of specialized types are probably more limited, although certainly flat arrays of types like `InlineOptional<InlineByte>` are highly desirable. In any case, the ability to create specialized array types is decoupled from the bytecode design by funnelling the creation of specialized arrays through an upcall to the associated species object.

These upcalls will be bypassed (short-circuited through JVM logic) for cases which the JVM already knows are "hardwired". For example, type tests of a species against its "raw" default species or its super-species are hardwired this way.

It may be that not all of these upcalls will be needed. Further experimentation will guide us.

Bi-variant specialization linkage

A bi-variant method *M* in *C* is one where *C* is parametric over some *R*, and *M* is *also* parametric over some other *Q* of kind `PARAM_MethodAndClass`. A call site for such a method can refer to a constant which is comprised of all of these elements:

- The "raw" reference to *C*, a `CONSTANT_Class` constant.
- Optionally, a parametric reference to *C*, a `CONSTANT_SpecializationLinkage`
- wrapping the "raw" reference to *C*, and also proposing some
- linkage parameter value *V*. Call this the "scope wrapper" if it's present.
- The "raw" name and type of *M*, encoded in a `CONSTANT_NameAndType` constant.
- A `CONSTANT_Methodref` (or `CONSTANT_InterfaceMethodref`) which refers to both the reference to *C* (whether parametric or not) and the name and type of "M".
- Optionally, a "top level" parametric reference to *M*, a `CONSTANT_SpecializationLinkage` wrapping the previous reference to *M* (whether "raw" or not), and also proposing some linkage parameter value *W*. Call this the "member wrapper" if it is present.

It is thus possible that a parametric reference can propose *two* linkage parameters, one to be validated on the anchor *R* on the containing type *C* and the other to be validated on the "inner" anchor *Q* for *M*.

A simpler reference might propose just one of the two proposed anchors, using either the scope wrapper or the member wrapper. The simplest possible reference omits both wrappers; this would be a completely "raw" (but still legitimate) reference to *M*.

If the reference to *C* contains a scope wrapper, *V* is validated (for *R*) during the parametric resolution of *C*, without reference to *M*. The resulting `SpecializationAnchor` value is *V1*.

If there is no scope wrapper (the reference to *C* is "raw"), then the default specialization (for *R*) is obtained as *V1* as part of the resolution of the reference to *C*, again without reference to *M*.

If the reference to *M* has no member wrapper, then the previously validated value *V1* is proposed as the linkage parameter for *M*. If *M* were simply parametric (over *R* again), this would be exactly correct. But since *M* (in this scenario) is bi-variant, the proposed value *V1* fails to validate for *Q*.

Instead, as a special case, the JVM automatically derives from *V1* (validated for *R*), an *inner default* for *Q*. This is a default specialization for *Q* within the outer specialization *V1*.

This *inner default* was already created when *C* was prepared, as the regular default for *Q*, in the case where *V1* is the default for *R*. But even if *V1* is not the default specialization, the JVM must prepare and record an inner default for *Q* which is specialized within *V1*. This preparation must occur at most once, lest there appear to be multiple "raw" `PARAM_MethodAndClass` method specializations within some type specialization.

An implementation can eagerly prepare all possible inner defaults for an anchor of kind `PARAM_Class` when the outer `SpecializationAnchor` is prepared. Alternatively, it can prepare them lazily as needed. In any case, the metadata for a specialization of kind `PARAM_Class` should reserve space to link to each default ("raw") specialization of kind `PARAM_MethodAndClass` that might be built within it.

Because the JVM can automatically derive an inner default for *V1*, then there is no need for a second bootstrap method call. The inner default for *Q* (within any *V1*) is by definition valid for *Q*.

Finally, the reference to *M* can have a member wrapper which proposes a second linkage parameter *W*, to be applied in the presence of *V1*. In this case, it may be that *W* is already a valid `SpecializationAnchor` for *Q*, in which case *V1* can be ignored and the resolution is complete.

(We could mandate an error check to detect if *V1* is not the enclosing specialization when *W* is valid for *Q*. This seems not worth the effort, as it would detect only minor irregularities in the shape of the `class` file, which do not entail dangerous type errors.)

Otherwise, the inner default for *Q* within *V1* is obtained (as in the earlier case of a missing member wrapper), and supplied, along with the unvalidated *W*, to the bootstrap method for *Q*. Although *V1* is not directly passed to the bootstrap method, it may be readily obtained, since it is the outer specialization for *Q*, and the inner default is passed as the argument *B0* to the bootstrap method.

Volume III: Parametric Bytecode Instructions (JVMS-6.5)

Review: "Nominal" bytecodes and symbolic references

Some bytecode instructions can refer symbolically to API points. These instructions, sometimes called "nominal" bytecodes, contain the index (in their associated constant pool) of a symbolic reference to an API point, a class, interface, method, or field (depending on the particular bytecode). These instructions are:

- `ldc` and `ldc_w` may resolve a symbolic reference to a class or interface (via a `CONSTANT_Class`) or a symbolic reference to a field, method, or constructor (via a `CONSTANT_MethodHandle`)

- The `getfield`, `putfield`, `withfield`, `getstatic`, and `putstatic` bytecodes (which may collectively be called *access bytecodes*) resolve a symbolic reference to a field, via a `CONSTANT_Fieldref`.

- The `invokestatic`, `invokevirtual`, `invokeinterface`, and `invokespecial` bytecodes (which may collectively be called *invocation bytecodes*) resolve a symbolic reference to a method via a `CONSTANT_Methodref` or `CONSTANT_InterfaceMethodref` constant.

- The `new`, `instanceof`, `checkcast`, `anewarray`, `multianewarray`, and `defaultvalue` bytecodes (which may collectively be called *type-using bytecodes*) resolve a symbolic reference to a class or interface via a `CONSTANT_Class` constant.

The first execution of a nominal bytecode instructions of any kind generally entails resolution of its symbolic reference. The resolution is stored permanently in an associated constant pool state for the symbolic reference constant, and used for all executions of the instruction.

If two or more bytecodes share a single constant pool entry, they also share the associated resolution state. This sharing does not hold true for the non-nominal `invokedynamic` instruction.

Thus, nominal bytecodes resolve *API point references* (as defined above) to *API points* (also defined above). Since API point references can be parametric, it follows that nominal bytecodes may also be parametric. The behavior of such bytecodes is modified by the extra information in the specialization determined during resolution of a parametric API point reference.

All nominal bytecodes can be parametric

The bytecode instructions of a method may refer to a parametric constant *A* over some anchor *R* only if the method itself is parametric over *R*.

Therefore, if a method is not parametric at all, its bytecodes can only use invariant constants. Recall that all `CONSTANT_SpecializationAnchor` constants are parametric, but a `CONSTANT_SpecializationLinkage` constant may be either parametric or invariant, as determined by its specific dependencies.

In the context of bytecode execution of a method which is parametric over some anchor *R*, this anchor is *bound* to `SpecializationAnchor` object indirectly requested by a linkage parameter supplied by the caller. This contextual specialization is permanent for the duration of the stack frame. If the method is parametric over two anchors (bi-variant), both anchors are contextually bound; in fact the inner anchor uniquely determines the specialization of the outer anchor.

Wherever some *R* is bound, if *R* depends on some other anchor *Q*, *Q* is bound also. That can only happen if *R* and *Q* are of kinds `PARAM_MethodAndClass` and `PARAM_Class` respectively. In the future, if additional nesting modes are made available in class files, additional dependencies between anchors may become possible, and more complex simultaneous specializations may appear, reflecting multiple levels of scoping or nesting defined by one constant pool.

The following bytecode instructions potentially interact with parametric constants:

- The `ldc` bytecode (as well as `ldc_w`) may refer to a parametric constant of tag `CONSTANT_Class`, `CONSTANT_MethodHandle`, `CONSTANT_Dynamic`, `CONSTANT_SpecializationAnchor`, or `CONSTANT_SpecializationLinkage` (the last two are new tags, for a new kind of loadable constant).

- The `ldc2_w` bytecode may (as type-appropriate) may refer to a parametric constant of tag `CONSTANT_Dynamic`.

- The `getfield` bytecode (as well as the other access bytecodes `putfield`, `withfield`, `getstatic`, and `putstatic`) may refer to a parametric constant of the new tag `CONSTANT_SpecializationLinkage`, as well as a plain (or perhaps parametric) `CONSTANT_Fieldref` constant.

- The `invokestatic` bytecode (as well as the other invocation bytecodes) may refer to a parametric constant of tag `CONSTANT_SpecializationLinkage`, as well as a plain (or perhaps parametric) `CONSTANT_Methodref` or `CONSTANT_InterfaceMethodref` constant.

- The `new` bytecode (as well as `instanceof`, `checkcast`, `anewarray`, `multianewarray`, and `defaultvalue`) may refer to a parametric constant of tag `CONSTANT_SpecializationLinkage`, as well as a plain `CONSTANT_Class` constant.

In all cases, the resolution of the referenced parametric constant depends on the contextual specialization anchor, which in turn determines the preparation of the resolution state of each constant that depends on it.

Note that both old and new tags can be parametric. Conversely, a constant with new tag `CONSTANT_SpecializationLinkage` may be invariant, if it depends only on invariant component constants. A constant with new tag `CONSTANT_SpecializationAnchor` is always parametric; indeed such constants are the source of parametricity in all other constants. Any bytecode that uses a `CONSTANT_SpecializationLinkage` constant validates a proposed linkage parameter for the indicated API point, and uses the resulting specialization anchor as part of its execution, as described below. This is true whether that `CONSTANT_SpecializationLinkage` constant is itself parametric (e.g., `List<T>.get` for some local type `T`) or invariant (e.g., `List<InlineInt>.get`). Any invocation bytecode that uses a `CONSTANT_SpecializationLinkage` constant (whether parametric or invariant) to invoke a parametric callee method influences the selection of that callee’s specialization anchor.

Method Invocation

The bytecode instructions `invokestatic`, `invokespecial`, `invokevirtual`, `invokeinterface`, and `invokedynamic` are collectively called “invocation instructions”. All of them encode their stack effects by means of a descriptor string which contains descriptor types for arguments and return values. All but the last are so-called “nominal instructions”, which incorporate a symbolic reference to a method (or constructor) as the target of the invocation.

The execution of any invocation bytecode presupposes correctly typed arguments already pushed on the stack. It pops these values (if any) and passes them to a receiving method. The receiving method, if it does not terminate with an exception, will return any result value (as required by the descriptor), and the invocation bytecode will finish with that result value (if any) pushed on the stack.

These stack effects are strongly typed according to the JVM’s descriptor type system, as enforced by the verifier. Note that all such verified types are invariant; they are not affected at all by parametricity. Thus, although a parametric method may logically work with arguments or return values of parametrically defined types, it will physically use an invariant supertype (typically a type parameter bound), as encoded in a descriptor string, to describe the stack effects of the method. The verifier, which is plenty complex already, is mercifully ignorant of parametric effects.

Execution of a nominal invocation bytecode starts by resolving the symbolic reference to determine a specific method (or constructor) to execute. The resolution of the API point is unaffected by the presence of parametricity. In particular, a symbolic reference is always invariant, a hard-coded name and type, and located in a named class.

For simplicity, we are not extending symbolic resolution to locate methods in variant types such as species. The parametricity mechanism is cleanly separated from the complexities of the JVM’s class and interface type hierarchies and their members. It may be natural to allow type species some activity where today we only allow classes and interfaces; this is TBD for now. We are certainly not allowing variance in method signatures; this would greatly explode the complexity of every part of the JVM that needs to understand type descriptors and signatures.

If the operand of a nominal invocation bytecode is a `CONSTANT_SpecializationLinkage` constant, the symbolic reference and the associated linkage parameter are resolved, in that order. During resolution, if the method is parametric, the linkage parameter is validated and the resulting specialization anchor is permanently recorded with the resolved symbolic reference. This anchor is then passed, along with the arguments, to the callee method, for every invocation.

Some nominal invocation instructions perform “virtual method invocation”, which incorporates an extra method selection step to replace the resolved method (from the symbolic reference) by an overriding method. The overriding method has the same name and type descriptor, and is always concrete, but maybe be defined by a different class or interface, and (crucially) may be parametric. (If it is parametric, it will be so in a different `class` file from the class file of the overridden method.) The rules for this are complex and are described elsewhere.

The opposite of virtual method invocation is “direct method invocation”. Each invocation instruction performs virtual or direct invocation, based on its kind and result of resolution:

- `invokeinterface` invocation is virtual.
- `invokevirtual` invocation is virtual unless the resolved method is `final`, in which case it is direct.
- `invokestatic`, `invokespecial`, and `invokedynamic` are direct.

(Note that method handle invocation is direct, since the signature polymorphic invocation methods are `final`. The method handle may internally perform additional method invocation of one or more target methods. Since `invokedynamic` performs a method handle invocation, it is also direct.)

When a parametric method is invoked directly, the relevant specialization anchor is passed directly to the callee, and becomes available to execution in that callee’s stack frame, if the callee is in fact a method with a bytecode attribute (neither `abstract` nor `native`).

The processing of specialization anchors during `abstract` and/or virtual method invocation will be discussed in its own place later on.

Parametric Method Execution

Every parametric constant is resolved from a constant pool state that is associated with the validation of its associated (inner-most) specialization anchor. Thus, if a method is not parametric, its bytecode instructions must not resolve any parametric constants. If a method is parametric over some constant *R*, it may not resolve any parametric constant which is *not* also parametric over *R*. In both cases, a method bytecode attempting to resolve an inappropriately parametric constant will complete abnormally with a subclass of `LinkageError` (TBD).

(Inappropriately variant constant references could also be checked earlier, in the verifier. This does not seems to confer any performance benefit on the JVM, and the verifier is already complicated enough, so we won’t burden the verifier with this chore. Compilers can catch their own bugs without the JVM’s help on this. Because it works on “raw” JVM type descriptors, the verifier is blissfully oblivious to the effects of specializations.)

Every stack frame (§2.6) in the Java virtual machine contains an associated reference to the run-time constant pool (§2.5.5) of the class of the current method. (This is true in all versions of the JVM.) If the method is parametric over some specialization anchor *R*, this reference to the constant pool contains two component references, one to (the resolution states of) the invariant constants, and one to (the resolution states of) those constants which are parametric over *R*.

The resolution state of the latter constants is tracked distinctly for distinct specializations of *R*. Thus, the second component reference, to the parametric constant states, may vary from invocation to invocation of the method. This reference, however, is constant in any particular stack frame, and across any call chain that starts with some invariant `CONSTANT_SpecializationLinkage` that selects *R*, and continues to use the same specialization *R* for callees.

The first time a parametric constant *C* is resolved, its specialization anchor *R* (co-parametric with *C*) is consulted. The anchor *R* contains (along with other data) a set of resolution states for all *R*-variant constants, including *C*. As with an invariant constant, *C* is resolved in terms of its own structure and the constants it depends on. Since *C* is *R*-variant, its resolution can query the value of *R* or other *R*-variant constants, and so *C*’s resolution makes use of *R*’s associated resolution states and other data (such as a validated linkage parameter value). When *C* is resolved, the result of the resolution (whether normal or erroneous) is recorded in the same associated resolution states within *R*. Further resolutions of *C* relative to *R* produce the same result.

Later on, if the same method is called with a specialization anchor *R2* which (though derived from the same class file constant) is different from the previous *R*, *C*'s resolution state will be unaffected from any of the outcomes described in the previous paragraph, since the resolution states of *R* and *R2* are unrelated.

It is the responsibility of the bootstrap method of *R* to decide when and whether to create fresh specialization anchors for *R*-variant constants, or whether to find and reuse pre-existing specializations, with their pre-existing resolution states.

It is legitimate to return fresh specializations every time from the bootstrap method; in that case, if the caller of the *R*-variant method records their anchors in a `CONSTANT_SpecializationLinkage` state, then specializations are reused in the *R*-variant method only for calls from the same caller. This can enable a level of customization for a whole static call tree, independent of JIT inlining decisions.

Note that a method can be parametric over any kind of anchor, even the `PARAM_Class` kind. Although it seems odd to give a class anchor to a method, it is the most natural and efficient thing to do, in the common case where a class has just one specialization anchor (representing one group of type variables) shared everywhere. In particular, the constructor of a class (which for a primitive class is really a static factory) should be co-specialized with the class, so that specializations can be computed at call sites and then used (unchanged) by the `new` instruction (or `defaultvalue` instruction) which creates instances to be initialized by the constructor, and/or which creates instances within a factory method.

Note also that if a method does not make use of a class anchor, either in its type restrictions or in its body, it should be declared invariant in its `class` file, even if the source-level type parameters were in scope. If the method is overridden by another method which uses another anchor (as declared in a subtype), the linkage parameter may be loaded from the instance, if only reflective use is needed.

The anchor kinds besides `PARAM_Class` are expected to be useful for driving type information in complex parametric algorithms such as `Arrays.sort`, where there is no object instance to act as a direct "witness" to types or other contextual information. Even if there were only `PARAM_Class` anchors, the JVM would be required to treat them (for some API points) identically to non-`PARAM_Class` anchors, starting with parametric constructors and factory methods. Given the need to plumb such pathways, supporting algorithms like `Arrays.sort` is simply a matter of decoupling the JVM's legitimate need to associate specialization with instances (`PARAM_Class`) from its equally legitimate need to associate specialization with methods, including situations where there is no class specialization in sight.

The third anchor kind arises from this factoring by an observation that such split specialization scopes arise, sometimes, in source code, and can be supported by a modest incremental JVM effort.

Class specialization anchors are heavyweight compared with non-class anchors, because they record object layouts and other schema information. Non-class anchors amount to small heap objects that carry around type tokens and associated resolution states (as needed), and perhaps point to an enclosing class specialization.

Instance Creation

The `new` and `defaultvalue` instructions are called "instance creation instructions".

The `new` bytecode creates a new object instance, in a blank state, to be completed by a direct call to a constructor (`<init>` method). Its operand is a `CONSTANT_Class` constant which directs which class to create. The verifier ensures that each execution must be coupled (along every non-exceptional path) with exactly one invocation of a constructor of the same class.

A `defaultvalue` instruction does the same, except for an primitive class. It typically executes inside of a static factory for the class. The details of the two instructions are different, but the operand processing is the same, when the class is parametric.

The operand of a `new` or `defaultvalue` bytecode may also be a `CONSTANT_SpecializationLinkage` constant which wraps a `CONSTANT_Class` constant. Such a bytecode is called a "parametric instance creation instruction".

Much as with method invocation, when the JVM executes a parametric instance creation instruction, it first computes a class specialization and then applies it to the creation of the instance of the resolved class.

The kind of the specialization anchor must be `PARAM_Class`, and it determines the size and layout of the instance created, if the instances class (or any super class) contains any parametric fields.

If, conversely, an instance creation instruction is executed on a plain `CONSTANT_Class` constant, and the resolved class is parametric, then the default specialization (for that class) is implicitly used.

Due to layout customization, a highly optimized JVM might assign different sizes to different species of the same class. The sizing information in such a JVM is presumably stored on the validated `SpecializationAnchor` object of the `PARAM_Class`-kinded anchor.

If a supertype *S* (class or interface) of a class *C* is parametric, then the reference to *S* in the class file for *C* could have been either a "raw" `CONSTANT_Class` or else a wrapped `CONSTANT_SpecializationLinkage`. In either case, when *C* is loaded the reference to *S* is resolved and assigned its own specialization. When the instance of *C* is created, any fields defined by *S* are accordingly specialized, and may in fact participate in layout customization. This can happen even if *C* is *not* parametric.

The JVM ensures, as a global invariant, that every validated `PARAM_Class` anchor, for a concrete class, contains all required information about the size and layout of parametric fields declared in that class and all its super classes. This information is recorded by the JVM in association with every `SpecializationAnchor` object for an anchor of kind `PARAM_Class`, and specifically for each parametric non-static field in the class and in each of its parametric supers. See discussion about "f-tables" below.

For the `new` instruction, as with other possibly-parametric instructions, the verifier ignores the `CONSTANT_SpecializationLinkage` wrapper as if only the wrapped `CONSTANT_Class` constant were present.

The new instance, in turn, is permanently associated with the specialization anchor object. Note that the `new` instruction may be executed many times, but the resolution step (which computes and records the anchor) happens just once, before the first instance is created.

These rules ensure that the new object immediately "knows" which species it belongs to, just as all objects "know" which class they are in. This opens the door for JVM implementations to aggressively customize the layout of the new object for its particular species.

It is also a consequence of these rules that all instances created from the same `CONSTANT_SpecializationLinkage` constant state will share a common species. For a generic factory method, this means that every distinct species of C created by the method is logically associated with (at least) one `CONSTANT_SpecializationLinkage` constant state. The actual bookkeeping for this state is organized so that each monomorphic caller of the factory remembers the associated linkage state, and the factory method can use common code (and/or customized code) to perform the instance creations. These linkage states are designed so as to allow lightweight implementations in the JVM, while still supporting significant optimizations when desired.

Parametric field types (i.e., sizes and layouts) are determined by the bootstrap method call that originally produced the validated species of C that is applicable to a particular `new` or `defaultvalue` bytecode execution. See the section below on specialized types.

The Java language encourages constructors to perform a series of `putfield` (or `withfield`) instructions on each fresh object instance. This is especially true for blank final instance variables. Fresh instance creation, constructor invocation, and field access can all be parametric operations, both individually and in larger cooperative patterns, defined by a translation strategy.

Note that a specialized parametric field will incorporate a runtime check (as if by `checkcast`) that can enforce a specialized type restriction on the stored value. This is true even if (for some reason) the constructor is unspecialized, or is invoked with the default "raw" specialization and passes that specialization to its `putfield` or `withfield` instructions.

As with non-parametric `new` instructions, the blank new instance created by a parametric `new` instruction will be unusable until a corresponding call to the constructor has completed. But the verifier does not track specializations at all, and so has no role in aligning the specialization of the `new` bytecode and the subsequent `invokespecial` of a constructor. (A similar point can be made about a `defaultvalue` bytecode and subsequent `withfield` operations.) Thus it is possible that the constructor invocation after parametric `new` instruction will use a different specialization, or none at all. (A `new` instruction for a non-parametric class might also be followed by a parametric constructor invocation, useless as this would seem.) The JVM makes not attempt to validate the overall consistency of specializations in such code shapes, leaving them to translation strategies to define and enforce, on top of the dynamic effects of individual bytecodes.

The rules for checking and using specializations are defined by the JVM on the basis of single instruction executions, not on the basis of larger bytecode patterns which the verifier might enforce. This choice is made because the verifier is a relatively poor way to ensure general well-formedness of bytecodes; it should only be used in those rare circumstances where there is some proven need to improve interpreter performance.

Array creation

The array creation instructions are `anewarray` and `multianewarray`. They both accept `CONSTANT_Class` operands, and can also accept species operands, in the form of `CONSTANT_SpecializationLinkage` operands which wrap `CONSTANT_Class` operands.

As has always been the case, the operand of the array creation instruction is first resolved, and may resolve to a species or (in the case of `multianewarray`) into an array of species type. The appropriate array type is determined and instantiated. If a species supports reified array types, that is determined by the runtime support.

The resolution of `CONSTANT_SpecializationLinkage` constants in the presence of array type descriptors has been glossed over so far, but is simple to specify: First the element type of the array is resolved, and then, if it is parametric, the proposed linkage constant is resolved and validated. (This is just as if the array type descriptor had not been present, but instead the plain element type name were the subject of the `CONSTANT_Class` constant.) Once a species is determined, an upcall to the species reflector object determines an array type that will contain it (as a `Class` object). The array creation instruction then makes use of that array type to build the required array. Note that because the verifier does not track species, only a cast to the "raw" array type is needed to maintain correct types in the bytecode.

Field Access

The bytecode instructions `getfield`, `putfield`, `withfield`, `getstatic`, and `putstatic` are collectively called "field access instructions". As with invocation instructions, they encode a symbolic reference to a class member (a field not a method). They also accept `CONSTANT_SpecializationLinkage` constants which wrap their symbolic reference.

The JVM keeps careful track of the layout of each specialized class. When accessing parametric fields within that layout, it concentrates on implementing one particular fast path, the path that occurs when the field access instruction uses exactly the same specialization as the object instance's class *C*.

The JVM ensures that if two instances have the same species, then their layouts are completely compatible; in particular the type restrictions are the same. Normally, if two instances have the same species, their class specialization anchors are identical, but in some cases this may not be the case. Though it is possible to create two class specialization anchors with a common species, it is impossible for the two specializations to differ in their layouts. (The factory methods for specialization anchors ensure this.)

In particular, the object instance's species (*C* with a class specialization anchor *R*) is examined to see if the object was created with the identical species as is being proposed by the field access instruction (after resolution). If the match is exact, then the JVM can confidently access internal layout and type information and load the specialized field.

- For `getfield`, the specialized value is loaded and then (if necessary) cast to the unspecialized type in the field's symbolic reference.
- For `putfield`, the unspecialized value is popped from the stack, and then cast (if necessary) to the specialized type and copied into the identity object.
- For `withfield`, the unspecialized value is popped from the stack, and then cast (if necessary) to the specialized type and incorporated into a new version of the primitive object.

If the field is in a superclass *S* of the instance class *C*, the JVM checks for the fast path by matching the species of the field reference constant (which can "see" *S* despite mentioning only *C*) to the corresponding supertype species declared by *C* (assuming *C*'s anchor *R*, if relevant).

Since the superclass is parametric, the symbolic reference in *C* to its super *S* must either be an invariant constant (either a `CONSTANT_SpecializationLinkage` or a "raw" `CONSTANT_Class`) or a parametric `CONSTANT_SpecializationLinkage` constant depending on *C*'s anchor *R*. In either case, the instance type *C* (accompanied by a species object from *R* if *C* is parametric) must always determine an associated "push up" spcies for the super *S*. This latter species is the subject of the fast path check when the field is inherited.

There is also a slow path which is used when the layouts do not match. This is used to implement raw access, and perhaps other type relations between specializations. Other than support for raw access, it is TBD.

Type checking

The bytecode instructions `instanceof` and `checkcast` are called "type checking instructions". Both of them refer to constant pool entries which represent the types being checked. Both of them accept `CONSTANT_SpecializationLinkage` as well as `CONSTANT_Class` constants.

When their operand is a class specialization, they operate on the species associated with that specialization.

The constant is resolved in either case. The following cases apply:

- The constant resolves to an array type, with either a parametric or non-parametric component type. In that case, the array type is directly checked, as resolved. (If the specialization did not "refine" the actual array type, arrays of different specializations might be confused. This issue is TBD.)
- The constant resolves to a non-parametric class or interface, The behavior is unchanged from previous versions of the JVM. (One difference: The symbolic reference is resolved even if the stacked operand is `null`.)
- The constant resolves to a parametric class or interface (with the default specialization or some other specialization). In that case, the instruction first checks the class or interface as if no specialization were present, and then checks the species against the corresponding species recorded in the object instance. (See discussion of "s-tables" below.) As a special case, if the corresponding object species is a "raw" default specialization, the check succeeds, because "raw" objects are welcome everywhere.

In the last case, there is a fast path and slow path. If the species linked into the type check instruction is identical to the species recorded in the instance, the check succeeds.

Otherwise, the slow path is taken. It is possible to design an upcall to the species object recorded in the constant pool for the resolved operand, allowing a certain amount of user programmability.

As a special rule, if a `checkcast` instruction refers to a `CONSTANT_SpecializationLinkage` constant instead of as `CONSTANT_Class` constant, the constant is resolved *even if the stacked operand is null*.

This allows the enhanced `checkcast` logic for primitive objects to reject null references. Recall that even if a primitive class is invariant, it is legitimate to refer to it via a `CONSTANT_SpecializationLinkage` constant; the proposed linkage parameter is simply ignored.

Somewhat uncomfortably, the constant pool does not distinguish between a parametric type `Foo<?>` and the "raw" instance of that type `Foo<raw>`. In fact, it uses `Foo<raw>` (with the built-in default specialization) as the meaning of all "raw" `Foo` symbolic references. This ambiguity causes some pain with type checking instructions, because if `Foo` is shorthand for `Foo<raw>`, you can only ask one of the following two questions: (a) Is an object *X* an instance of any species of class `Foo`? Or, (b) is an object *X* an instance of the default specialization (the "raw" one, not any more specialized one) of class `Foo`? For now we suggest papering over this distinction by asking the species object for `Foo<raw>` to serve double duty as `Foo<?>`, by recognizing, even if only via a slow path, all specializations of `Foo`, and not just the instances which pass the fast path. This means that some other idiom must be used to classify instances of the "raw" specialization, such as (hypothetically) `x instanceof Foo && x.getSpecies().isDefault()`.

The relations between API points and structures which refer to them (both old and new) are summarized in Diagram 4.4-F(a).

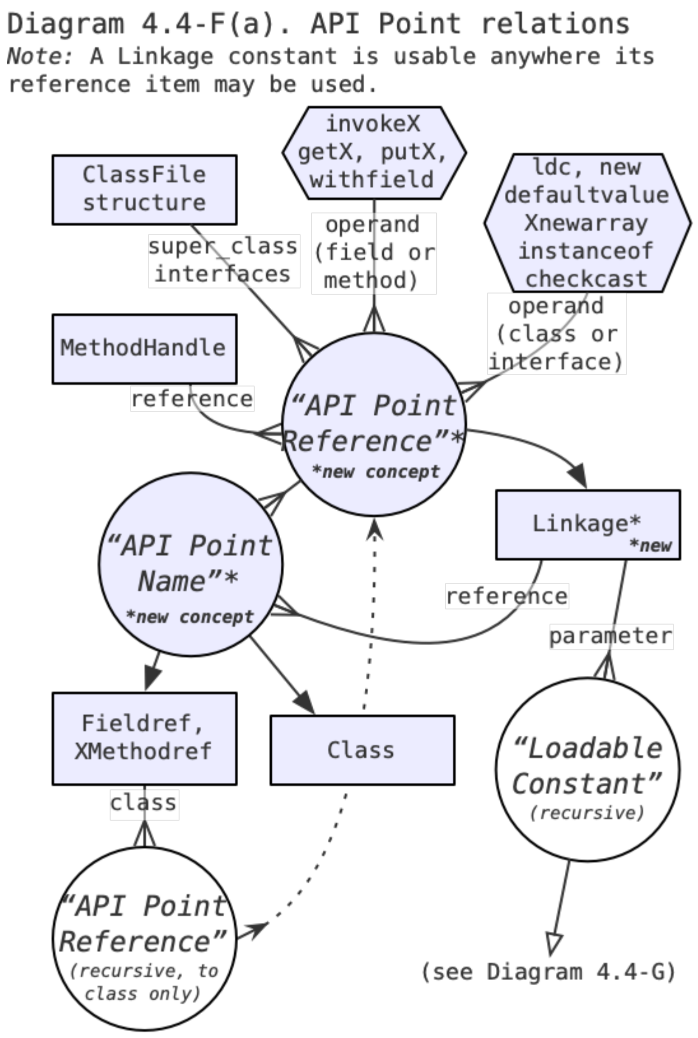


Diagram 4.4-F(a). API Point relations
Note: A Linkage constant is usable anywhere its reference item may be used.
invokeX getX, putX, withfield
ClassFile structure
ldc, new defaultvalue Xnewarray instanceof checkcast
super_class interfaces
operand (field or method)
operand (class or interface)
MethodHandle
reference
"API Point Reference"* *new concept
Linkage* *new
"API Point Name"* *new concept
reference
parameter
Fieldref, XMethodref
Class
"Loadable Constant" (recursive)
class
"API Point Reference" (recursive, to class only)
(see Diagram 4.4-G)

Volume IV: Virtual Dispatch and Calling Sequences

Apart from specialization, virtual method invocation is a relatively simple enhancement of simple direct invocation. Direct invocation is a simple relation from a caller to a statically resolved callee, embodied (usually) as a jump to a known function address. The caller and callee might agree exactly on the type declaring the method being invoked. Or, if the callee is declared by a supertype of the type mentioned by the caller, the lookup can be performed negotiated once, when the symbolic reference in the caller is statically resolved. For example, `MyNode::hashCode()` might resolve statically to `Object::hashCode()` if `MyNode` does not override the method from `Object`.

With virtual method invocation, the caller and callee can disagree about the receiver type, and the disagreement can be different for each execution of the call site. This is because the callee has two aspects: static and dynamic. The static identity of the callee depends on the symbolic reference at the call site, and is determined the same way as for a non-virtual call. For example, a call to `MyAbstractNode::hashCode()` might resolve statically to `Object::hashCode()` if `MyAbstractNode` does not override the method from `Object`, but if `MyConcreteNode` is a subtype of both, and it overrides `hashCode`, then a call site might statically resolve to `Object::hashCode()` while will choose a range of methods, including `MyConcreteNode::hashCode` (and `MyConcreteNode2::hashCode`, etc.).

The dynamic receiver type determines the dynamic callee method, by an extra process called called method selection. The effect of method selection is as if the call site were temporarily rewritten to mention the exact type of the receiver (for just this one call), and resolved from that very specific type, instead of the more general type mentioned statically by the caller. A call with a dynamic receiver thus breaks down, functionally, into two non-virtual direct calls.

Here are the steps to execute a single virutal call that requires method selection:

1a. Determine the caller's symbolic reference, including a receiver type *R1a* and a name and method descriptor.

1b. Using inheritance, determine a method declaration *M1* in some *R1b* (a super of *R1a*) which exactly matches the name and descriptor. Record information about *M1*, once per constant pool entry in the caller, as the static callee. (This is the last step, if *M1* has no overrides.)

2a. Determine the receiver's dynamic type *R2a*, which must be *R1a* or a subtype. (With interfaces there may be an extra cast to *R1a*.) Recall information about *M1* from step 1a.

2b. Using inheritance, determine a method declaration *M2* in some *R2b* (a super of *R2a*) which exactly matches the name and descriptor of *M1*. Invoke *M2* immediately.

Thus, when method selection is present, there are (for each dynamic call) four relevant receiver types, and up to two distinct methods. Luckily, the interactions between these moving parts can be partitioned.

For example, if the caller specifies the descriptor `MyNumeric::compareTo(Object)`, if `MyNumeric` (*R1a*) does not declare `compareTo`, then the static callee might resolve to the same method (*M1*) in `java.lang.Comparable` (*R1b*). Meanwhile, method selection on an instance of `MyInt32` (*R2a*) could select an inherited method `MyAbstractInt::compareTo(Object)` (*M2*, in *R2b* = `MyAbstractInt`).

In simple cases of method selection, we can add a small extra wrinkle to the basic action of jumping to a known function address: We look up a function pointer for *M2* in a known location relative to the dynamic receiver type *R2a*, and jump there. The famous "v-table" supplies this knokwn location for single-inheritance cases, and the computation of *M1* amounts to identifying a v-table offset, if the v-tables are set up with care (and they are). This simplicity begins to disappear when the dynamic type of the receiver is complex enough that the "known location" requires a search of several possible locations; this is (often) true in the case of multiple inheritance.

Specialization adds yet more complexity to the four receiver types and the relations between the methods *M1* and *M2*. Because of the structure of the parametric class file, there is no support for optional methods, and so the method selection process finds *M2* just the same with specializations as without. (That is the good news.) However, if the selected method *M2* is parametric, an appropriate specialization anchor must be computed for that selected method, and (in general) the class file for *R2b* (which declares *M2*) is independent of the class files for *R1a*, *R1b*, and *R2a*. Yet an anchor must be computed to feed to *M2*.

The amended calling process looks like this:

1a. Determine the caller's symbolic reference…

1b. Using inheritance, determine a method declaration *M1* in some *R1b*…

1c. If a `CONSTANT_SpecializationLinkage` constant is present, resolve a proposed linkage value *L1*. (This step 1c. can be done either before or after step 1b. A translation strategy has typically arranged the *L1* is appropriate to an API point in *R1a*.)

1d. If *M1* is parametric, compute a specialization for it by validating *L1* against *M1*'s anchor constant, to produce a specialization anchor *A1*. (This may need a bootstrap method call in *R1b*.) Record *A1* permanently in the constant pool of the caller, alongside the identity of *M1* itself. (This is the last step, if *M1* has no overrides.)

2a. Determine the receiver's dynamic type *R2a*…

2b. Using inheritance, determine a method declaration *M2* in some *R2b*…

2c. If *R2b* is parametric, recover the specialization anchor *A2* associated with *R2b* when the receiver (of type *R2a*) was created. (Thus, a virtual call may need to use specialization information associated with any super-type of the dynamic type of the receiver. See discussion of "s-tables" below.)

2d. If *M2* is parametric, compute its specialization anchor by jointly validating *A1* (the anchor for *M1*) and *A2* (the specialization for *R2b*). The validated result *A3* (for *M2*) may be cached, or it may be recomputed on every virtual call. If *A2* (for *R2b*) is already valid for *M2* it can be used as-is. (This will be true if *M2* and *R2b* are co-parametric.)

The joint validation step described above takes a specialization *A1* for *M1*, which might in general be a method specialization, and an independently determined class specialization *A2* for a super *R2b* of the receiver *R2a*. In general, the required result *A3* might itself be another method specialization, which is "inner" to *A2* (i.e., *A2* is sub-parametric to *A3*). If *A1* was also "inner" to some specialization *A0* of the static callee *R1b*, then the joint validation must apply language-specific rules to determine the value of *A3* such that the following ratio-like relation holds: *A0* is sub-parametric to *A1* just as *A2* is sub-parametric to *A3*.

Luckily, none of this needs to be encoded in the JVM specification.

Note that the class specialization for the dynamic receiver (*A2*) can be inconsistent with the static specialization anchor for the virtual call to *M1* (*A1*), due to heap pollution or similar effects outside the purview of the JVM. A bootstrap method in step 2d can enforce any language-level policy required concerning the various corner cases.

Parametric `abstract` methods

Although it would seem that `abstract` methods are mere placeholders, an abstract API (such as an interface) has a strong effect on specializations simply by declaring its abstract methods to be parametric. By declaring its dependency on an anchor, an abstract method forces a client's link resolution (against the abstract API, not any concrete implementation) to compute a specialization when using that API. That specialization is then made available to assist in specializing the eventual implementing method, if the latter method is also parametric.

The mapping of the specialization from superclass to subclass is probably language-specific, and in any case is (currently) the object of a runtime-supplied *virtual call connector* object. This object is created the first time a call site is executed (on a parametric `abstract` method), by means of an upcall (TBD). After this, every time a parametric method is selected, the JVM refers to the virtual call connector to supply a specialization anchor object for that method.

The virtual call connector may, at the language runtime's option, return the corresponding default specialization, extract an unvalidated "key" value from the `abstract` method's specialization anchor and revalidated that, or perform some other "pull down" mapping for the override method. It may (at the language runtime's option) cache the specialization anchor object, keying on the receiver class, its species, or some other value derived from the receiver. The cache may be kept local (a good thing since specialization information in the supertype is likely to be 1-1 with specialization information in the override), or it may be backed by some global table. The JVM stays out of the engineering details.

Parametric concrete non-`final` methods

The case of a parametric concrete method is essentially the same as for an `abstract` method, with the only difference being that the concrete method might be selected during a method call, if it there is no override (in the receiver class). The JVM has a fast path which, after the selection check, simply treates such a call as if it were a regular direct call.

If there is an override, the concrete method body is ignored, and the JVM's logic is the same as in the case of an `abstract` parametric method.

Thus, each invocation constant for a parametric method *M* has several possible components of state after resolution and invocation:

- The location of *M*'s declaration.
- A specialization anchor object for *M*.
- The identity of *M* (as a metadata pointer and/or v-table offset).
- A call connector for this call site (if a parametric override of *M* is called).

The need for call connectors can be reduced in by various expedients, and perhaps implementation of them can be deferred, by careful arrangement of translation strategy, at least early in prototyping. For example, perhaps types can be made parametric but methods can be invariant. This precludes customized calling sequences, but may be good for a start. If a supertype is parametric and its methods are parametric also, perhaps the overriding methods in subtypes can be made invariant. The hitch there is customized calling sequences may not be readily available if something prevents us from "copying them down" from the supertype. Also, if an overriding method happens to need access to a reified type, it will have to make a call to a synthetic API point defined (as a concrete parametric method) in the supertype.

Volume V: Specialized Class Layouts

Specialized class layouts are driven by the `anchor_index` items in `Parametric` attributes of parametric fields. This can be done automatically by the JVM in most cases, perhaps all.

If it is necessary for the runtime to give advice and consent on parametric field layouts, the natural place to do this is inside the bootstrap method upcall for a `PARAM_Class` anchor, *before* the call returns a fresh `SpecializationAnchor` object. Any special rules not encoded in `anchor_index` items can be injected into the creation of that object. In any case, this can be done reflectively, and so does not impose new requirements on the basic structure of `class` files, or the semantics of constants or bytecode instructions.

In a cooperating JVM implementation, a field type which is marked as `void` could be given a zero-byte presence in an affected class layout, as if it were an empty primitive class. Either tactic seems to lead to an efficient way to allocate a `boolean`-like flag field which is physically present in the layout of `InlineOptional<InlineLong>` but disappears in the layout of `InlineOptional<Object>`.

(More TBW, but see comments scattered throughout.)

Volume VI: Implementation Considerations

As a whole, this design attempts to push doubtful matters of language design and runtime implementation upward to the language runtime, via bootstrap methods and other upcalls.

When an upcall yields information to the JVM, there is a reliable specification of how and where that information is preserved. The presence or absence of optimization (at the JVM level) is no excuse for indeterminate behavior.

The splitting or lumping of type information into runtime species is a language decision. Once the JVM decides on a specialization, though, it treats it seriously and prepares constant pool states for resolving and recording constant pool entries and API points that are co-parametric with that specialization's anchor. The language translation strategy may take steps to minimize the number of such entries, but they are also probably inexpensive, on the order of one or two heap variables (machine words) per parametric constant pool entry.

Callers and callees perform handshakes to agree on specializations at every API point usage. If a specialization "splits out" a locally relevant species, it will be recorded at the call site, and its callees can develop their own dependent constants and recursive call sites, based on that recorded specialization and its resolution states. The result is that an invariant call to generic code creates a static tree of resolution states (prepared constant pool entries inside of specializations) which mirrors the dynamic shape of the generic code, including all of its generic subroutine calls, into and out of multiple `class` file artifacts.

For example, if `Arrays.sort` were made parametric, then similar static call trees (of `sort` and its helper methods) could be rooted at different user call sites, each call site specifying a different specialization: One for arrays of `Point`, another for `Color`, etc., where the array elements are primitive classes that are useful to specialize over, even customize over. Because specializations are associated with the resolution states of call sites, the call sites for sorting `Point` arrays will be kept distinct from those which sort `Color` arrays. In fact, the technique may help to refactor together the hand-maintained code which currently handles arrays of built-in primitives: Common code can (perhaps) handle not only (hypothetical primitive types) `Color` and `Point`, but also `int`, `long`, `float`, etc.

After a given generic call returns, if the call site is executed again, the static tree of resolution states is immediately available. If a method involved in this call tree becomes hot, the JVM can obtain good information on which constants might be useful to inline into customized code.

Data structures

Constant pool indexing
Resolution states
When an API point reference constant is resolved, the resulting state in the constant pool of the caller always includes a metadata reference to the resolved class, interface, field, or method.

In addition, the resolved API point is parametric, and if the API point reference specifies a linkage parameter that requests something other than a "raw" default specialization, then additional state information must be recorded in the caller to allow correct use of that API point.

These are the additional resolution states required to manage specialization:

- For a `CONSTANT_SpecializationAnchor` constant, a `SpecializationAnchor` object that reifies each of its specializations.

- For a `CONSTANT_SpecializationLinkage` constant (i.e., an API point reference), the validated `SpecializationAnchor` object obtained as part of the resolution of that constant's API point.

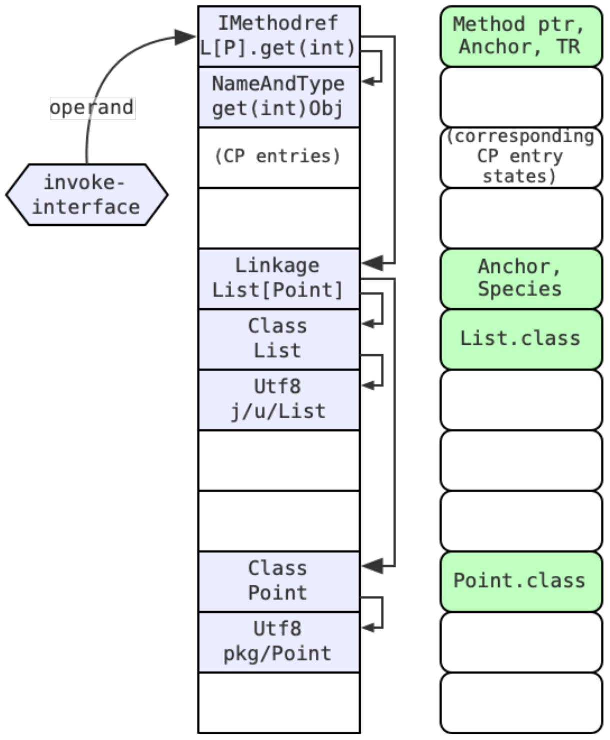
- For a `CONSTANT_SpecializationLinkage` constant which wraps a `CONSTANT_Class`, the validated `SpecializationAnchor` object must also supply species metadata for the specialized class or interface.

- For a `CONSTANT_SpecializationLinkage` constant which wraps a non-type API point reference (a `CONSTANT_Fieldref`, `CONSTANT_Methodref`, or `CONSTANT_InterfaceMethodref` constant), the the validated `SpecializationAnchor` object must also supply type restriction information for the specialized field or method.

- If used without a `CONSTANT_SpecializationLinkage` wrapper, a non-type API point reference may yet have its `CONSTANT_Class` API point reference wrapped a `CONSTANT_SpecializationLinkage`, and if the API point is co-parametric with its enclosing class or interface, the linkage state in the client must record a `SpecializationAnchor` object for the field or method, as well as any relevant type restriction.

An example constant pool, showing constants necessary for a hypothetical client of some interface species `List<Point>` to invoke a specialized method, along with a summary of their resolution states, is sketched in Diagram 4.4-H(a).

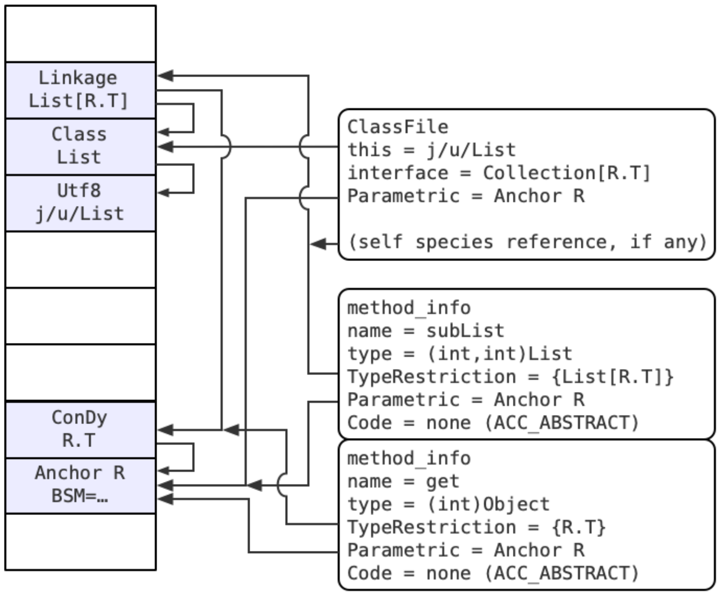
Diagram 4.4-H(a). Example constant pool:
non-parametric client of List<Point>
(resolution states are at right; all are invariant)



In this example, the resolution state for the `CONSTANT_InterfaceMethodref` for `List<Point>.get` has a `SpecializationAnchor` plus a type restriction for the `get` method (presumably requiring it to return a `Point` instead of a regular `Object`). The species reference `List<Point>` also stores a `SpecializationAnchor`, as well as a species descriptor for `List<Point>`. The `SpecializationAnchor` value is the same for both constants, assuming `List` and its `get` method are co-parametric. Note also that this class file has only invariant constants, although it will link to parametric API points in the class file for `List`.

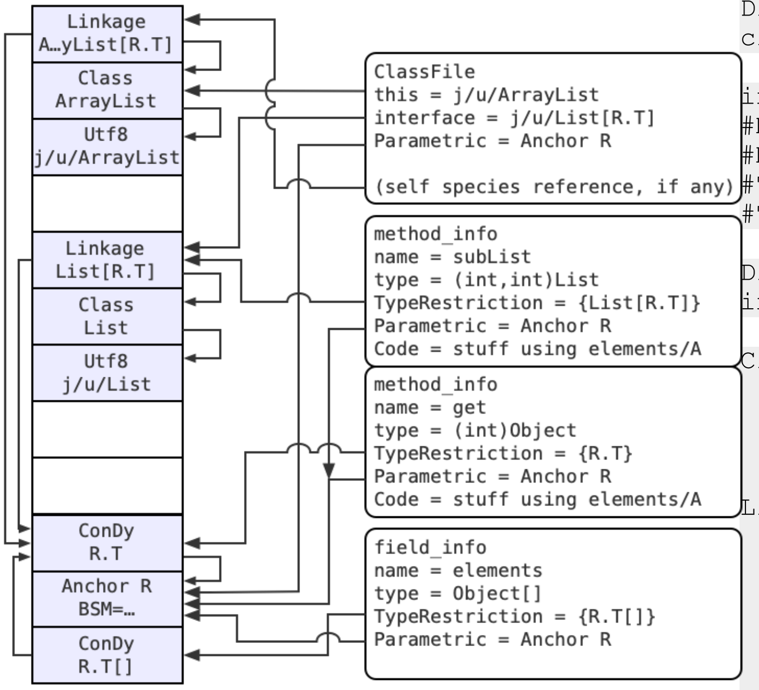
An example constant pool and class file structure for a hypothetical interface `List` which matches the client in the previous example is sketched in Diagram 4.4-H(b).

```
Diagram 4.4-H(b). Example parametric interface
    public interface List<T> { …get… }
```



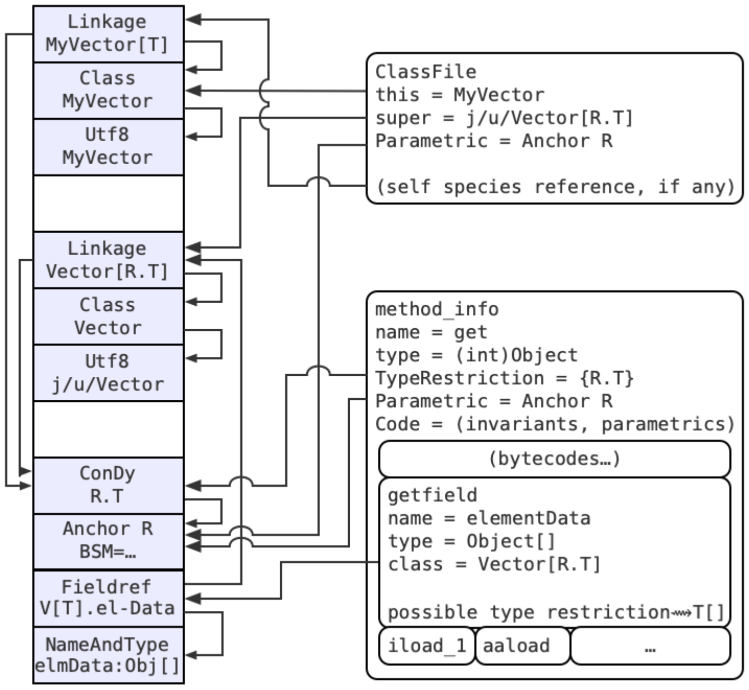
An example constant pool and class file structure for a hypothetical implementor `ArrayList` which matches the previous two examples is sketched in Diagram 4.4-H(c).

```
Diagram 4.4-H(c). Example parametric implementation
    class ArrayList<T> implements List<T> { …get… }
```



Here is another example constant pool, for a hypothetical subclass `MyList` of `java.util.Vector` that makes access to the inherited `elementData` field (hypothetically specialized), in sketched in Diagram 4.4-H(d).

```
Diagram 4.4-H(d). Example parametric subclass
  class MyVector<T> extends ju.Vector<T> { …get… }
  class Vector<T> { … protected T[] elementData; … }
```



Discarding the linear structure of constant pools, we can create a textual sketch of the same relations, as follows:

```
Diagram 4.4-H(a). Example constant pool:
client of List<Point>

invokeinterface #M56
#M56 = IMethodref[#T57.get(int)Object]
#M56.State = get:(int)Point in List<Point>
#T67 = Linkage[Class[List], Class[Point]]
#T67.State = species List<Point>

Diagram 4.4-H(b). Example parametric interface
interface List<T> { ...get... }

ClassFile {
  this = j/u/List
  interface = Linkage[Collection<#P83.T>]
  Parametric = #P83
  (self species reference, if any:
Linkage[List<#P83.T>])
  method_info {
    name/type = get/"(I)Ljava/lang/Object;"
    TypeRestriction = {#P83.T}
    Parametric = #P83
    Code = none (ACC_ABSTRACT)
  }
  #P83 = Anchor[BSM = make List species &
anchor]
  #P83.T = ConDy[extract T from #P83]
}

Diagram 4.4-H(c). Example parametric
implementation
class ArrayList<T> implements List<T>
{ ...get... }

ClassFile {
  this = j/u/ArrayList
  interface = Linkage[Class[j/u/List], #P42.T]
  Parametric = #P42
  (self species reference, if any: #T21)
  #T21 = Linkage[Class[ArrayList], #P42.T]
  method_info {
    name/type = get/"(I)Ljava/lang/Object;"
    TypeRestriction = {#P42.T}
    Parametric = #P42
    Code = {
      ...
      getfield #FR56 //
ArrayList<T>.elements:Object[]
      ...
    }
  }
  field_info {
    name = elements
    type = "[Ljava/lang/Object;"
    TypeRestriction = {ConDy[#P42.T[]]}
    Parametric = #P42
  }
  #FR56 = Fieldref[#T21, #NAT57]
  #NAT57 = NameAndType[elements, "[Ljava/lang/
Object;"]
  #P42 = Anchor[BSM = make ArrayList species &
anchor]
  #P42.T = ConDy[extract T from #P42]
}
```

```
Diagram 4.4-H(d). Example parametric subclass
class MyVector<T> extends ju.Vector<T>
{ ...get... }
class Vector<T> { ... protected T[]
elementData; ... }
```

```
ClassFile {
  this = MyVector
  super = #T20 //Vector<T>
  Parametric = #P51
  (self species reference, if any: #T19)
  #T19 = Linkage[Class[MyVector], #P51.T]
  #T20 = Linkage[Class[java/util/Vector],
#P51.T]
  method_info {
    name/type = get/"(I)Ljava/lang/Object;"
    TypeRestriction = {#P51.T}
    Parametric = #P51
    Code = {
      ...
      getfield #FR66 //
Vector<T>.elementData:Object[]
      iload_1
      aaload
      ...
    }
  }
  #FR66 = Fieldref[#T20, #NAT67]
  #NAT67 = NameAndType[elementData, "[Ljava/
lang/Object;"]
  #P51 = Anchor[BSM = make MyVector species &
anchor]
  #P51.T = ConDy[extract T from #P51]
}
```

Compact resolution states

A reasonable implementation strategy for recording resolution states is to store them in the Java heap, using plain `Object[]` arrays to store `SpecializationAnchor` objects, species (as mirrors), and type restriction information. Race-free update requires that each resolution state be patched in as a single word. Also, the initial null value in the array must represent an unset state, and there must also be provision for recording resolution errors. One way to meet these requirements would be a two-component `ResolutionResult` record which stores either a value or an error. For the normal case of a resolution value which is neither null, nor an error, or a particular `ResolutionResult` record, the value can be stored directly in the state array, without the overhead of a `ResolutionResult` record.

```
record ResolutionResult(Object value, Error
error) {
  public ResolutionResult {
    if (value != null && error != null) {
      throw new
IllegalArgumentException("cannot have both
value and error");
    }
    if (value == null && error == null) {
      throw new IllegalArgumentException("must
have either value or error");
    }
    // Note: The all-null state is reserved for
unresolved constants.
  }
  public boolean hasError() { return error !=
null; }
  public Object decode() throws Error {
    if (hasError())  throw error;
    return value;
  }
  public static Object decode(Object result)
throws Error {
    assert(result != null);  // caller
responsibility
    if (result instanceof ResolutionResult)
      return
((ResolutionResult)result).decode();
    return result;
  }
  public static Object encode(Object result) {
    return new ResolutionResult(result,
null).encode();
  }
  public static Object encodeError(Error error)
{
    return new ResolutionResult(null, error);
  }
  private boolean valueIsEnough() {
    // determine if the value itself is fully
informative (usual case)
    return error == null && value != null && !
(value instanceof ResolutionResult);
  }
  public Object encode() {
    if (!valueIsEnough())  return this;
    return value;
  }
}
```

Fast paths

Certain “fast paths” are directly executed by the JVM, without further reference to upcalls. The general rule is that a call site (or access site), as represented by a resolved constant pool entry, will execute at “full speed” after (at most) a check that a caller’s idea of a specialization is identical with a callee’s (or container’s) idea.

Graceful degradation on mismatch is also possible. (It is required, to support various roles of “raw” types with default specializations as legacy types and/or “wild” types.) The notions of virtual call connectors and virtual field connectors appear to supply a “fast enough” slow path, while allowing unpolluted code to speculate that such slow paths are, usually, irrelevant.

It might be possible to hardwire the default specializations even more into the JVM, so that every use point (over a non-default specialization) is bimorphic, allowing both default and (a single) non-default specialization. Maybe, maybe not. The number “two” is suspiciously close to the number “many” in software system design, and while “fast” and “slow” often coexist, “slow” tends to develop multiple purposes and aspects. Thus we propose the “connectors” as a way of implementing raw types as a language policy, rather than as a hardwired JVM behavior. If the only use of connectors is to implement raw types (unlikely but possible) the extra effort maintaining JDK code will still be balanced by the blissful ignorance of the JVM code, of the rules for raw types, even after paying for the plumbing of the connector APIs.

Fast revalidation
If a parametric API point *M* in some class *C* is linked with a proposed value which may already be a validated `SpecializationAnchor` for that API point, it is a matter of a few machine instructions to check for a fast path:

- If the value is in fact refers to a `SpecializationAnchor`.
- If the `SpecializationAnchor` is in fact for an anchor declared in *C*.
- If the `SpecializationAnchor` is associated with the correct anchor constant in *C*.

In that case, the linkage can be performed without invoking a bootstrap method. This is a common fast path when a class is linking to its own API points. It may also be a common fast path if translation strategies elect to expose `SpecializationAnchor` objects across classes.

A further fast test can detect if the anchor is for a default specialization, in which case no type restrictions will be present, and the call site may use legacy “raw” semantics.

Fast access to parametric constants
When a parametric constant *A* is resolved during method execution, there must be a specialization anchor *R* present, for the present method call, which contains a resolution state for *A*.

The interpreter must perform the steps such as the following:

- Determine which anchor constant *N* is co-parametric with *A*.
- Check whether the constant pool entry for *R* is *N*; if so, use *R*.
- If not, then constant pool entry for the outer specialization to *R* must be *N*. In that case, replace *R* by its outer specialization anchor.
- Find the table of resolution states in *R*.
- Determine the index *I* of *A* in that table. (This should be a statically assigned indexm, determined at class load time.)
- Inspect item *I* in *R*’s resolution state table.
- If item *I* is unresolved,, perform resolution logic and record the result.
- Then, if item *I* has a resolved value, use that value.
- Otherwise, item *I* has a recorded exception, so throw that exception.

When a constant pool is first parsed, a table can be built that classifies each constant, producing the following information:

- Which anchor is this constant co-parametric with (else 0)?
- What is the index of this constant in a compact numbering of it and its co-parametric constants?

These two values can help the interpreter quicklyl find split resolution states.

This compact numbering is similar to the compact numbering performed today for method and field references in HotSpot, in the so-called “constant pool cache”. It may be the techniques can be unified, so that the CP cache is really just a compact array of a certain population of CP constants, just as specialization state tables are a compact array of a different population of CP constants.

The resolved value of a `CONSTANT_Methodref` can be specialized, if its class link is specialized (i.e., is a `C_Linkage`). The metadata pointer (or index) can be stored in an invariant side table (the “constant pool cache”) but the specialization data for the class must be passed to the method call. This may need to be accessed from a split constant pool entry, not for the `CONSTANT_Methodref`, but rather for the `CONSTANT_SpecializationLinkage` that it refers to. A flag in the CP cache can give the interpreter a heads-up, to go hunting for this information.

Fast access to parametric fields: f-tables
When a class *C* is first loaded, the JVM must keep track of all its non-static fields, so it can compute a data layout for the class. Fields contributed by supers must also be tracked. It is useful to think of the JVM as making a table of all these fields, and to observe that each field has a unique position in this table. Even better, if a field *F* is inherited from a super *S* of *C*, the position of *F* can be contrived to be the same in the table for *C* as it is in the table for *S*.

If a non-static field is specialized with a type restriction which changes its layout (e.g., an inline primitive) or other access behavior (e.g., a cast), then this information must be tracked as well, and it differs in different specialization anchors. This information can also be thought of as being stored in a table with the same layout as the previous field table, except that it contains entries only for the parametric fields. This table may be called the “f-table”, by analogy with the well-known “v-table” which selects virtual methods; this f-table selects fields with virtualized semantics.

While the layout of an f-table may be settled statically when a class is loaded, its contents must be computed separately within every specialization of *C*. This implies that every specialization anchor links, somehow, to an f-table that provides the specialized field access behaviors.

What’s in an f-table entry? Well, there might be any number of things, depending on engineering decisions:

- metadata about the original field (or maybe that’s in a central table)
- the offset of the field in the specialized layout
- the specialized type restriction (if any) that applies to the field
- a format token describing the field’s format (as determined by the JVM)
- an optimized bit of code for reading the field
- an optimized bit of code for writing the field
- an optimized bit of code or data to process the field in the GC

When the JVM needs to read a specialized field, it takes these actions:

- compute (at resolution time) the f-table index *I*
- also compute (at resolution time) the expected species (raw or not)
- if the expected species is not raw, check-cast the instance

- find an f-table that matches the instance, and load item *I*
- use the item's offset to address the field
- use the item's format token (or reading code) to load the field

It is probably desirable that all invariant fields be located at fixed locations, reportable in the static field layout table created when *C* is loaded. That way the normal fixed-offset access methods can be used for these simple fields. It follows, then, that all specializable fields should come at the end of the instance layout (or maybe in available padding holes). This is true even if a parametric field is inherited from a super, and the sub-class contains invariant fields.

Fast parametric type checks: s-tables
When a class or interface *C* is loaded, the JVM must keep track of all the supers (super classes and interfaces) of *C*. Initially, these are resolved in unspecialized form, just as if no specialization were present. It is useful to think of the JVM as making a table of all these supers, and to observe that each super has a unique position in this table.

If a class or interface has specialized supers, those specializations are computed when the class or interface is prepared. (This must be after loading.) If a species has specialized supers, those specializations are computed when the species is prepared.

When a specialized super *S* of a class, interface, or species *C* is computed, the specialization anchor computed for *S* (during resolution) is recorded in a table of specialized supers for *C*, which may be called the "s-table". It is useful to think of the s-table as having the same layout and contents as the previously mentioned table of raw supers, except that each raw super is replaced in the s-table by its corresponding specialization anchor. An implementation can surely contrive to avoid allocating space for the non-anchor contents by appropriate numbering tricks.

When an object is queried (via *instanceof* or a type restriction) whether it is some raw type, the raw type table can be consulted, just as in the non-parametric case. When the query is against a species *S*, then the s-table can be consulted at the appropriate position (determined by the head type of the species *S*), and *S* compared against the species defined by the anchor *R* in the s-table.

One might think that the s-table entries should be species, instead of specialization anchors. That is reasonable, but note that the table of anchors is still needed for other purposes. Notably, when a class or species *C* is asked to resolve a member which is inherited from some super-species *S*, the resolution logic must be ready to apply the correct specialization anchor to the member of *S*, if it is co-parametric with its declaring class *S*. Which anchor is that? It is simply the anchor that was first computed when the link from *C* to the super *S* was resolved when *C* was prepared. That may be identical with the anchor species for *S*, or it may also include some private parametric data that is not reflected in the species per se.

When method selection is performed during virtual method invocation, s-tables can also be used to derive the correct specialization anchor for teh selected method. Alternatively, a v-table structure could be enhanced to hold not only method metadata references, but also the anchor corresponding to each such metadata reference.

These table structures can be engineered in many different ways. The layouts of the tables can be carefully tuned relative to a hashing or indexing scheme, or organized randomly and searched by linear search, or a combination of the two. The new requirement for specialization of supers is that such tables need to hold specialization anchors, as well as regular "raw" classes and interfaces.

It may be fruitful to adapt v-table layouts to hold s-table contents as well. In any case, a selectable method might hold an index or other key to retrieve its corresponding anchor from the v-table it was selected from, or else a nearby but distinct s-table.

Sample bootstrap API

```
package java.lang;

public sealed
interface Species
  permits Species.Impl
{
  /** The head type of this species.
   *  This is also the "raw" default layout and
behavior of this species.
   */
  Class<?> head();

  /** An object which the language
   *  runtime deems to be the validated
representation
   *  of a linkage parameter that could produce
this species.
   *  Typically a list or tuple of reflected
type arguments.
   *  Always returns {@code null} for default
specializations.
   *  Specialized fields should depend only on
this value.
   */
  Object parameters();

  /** Whether this object is a default "raw"
specialization,
   *  automatically created by the JVM.
   */
  boolean isDefault();

  /** A specialization anchor which defines
this species.
   *  Note that a species can be refined by
multiple
   *  specializations, so species can be one-
to-many
   *  relative to specializations.
   */
  SpecializationAnchor specialization();
}
package java.lang.invoke;

public sealed
interface SpecializationAnchor
  permits SpecializationAnchor.Impl
{
  /** Whether this object is a default
specialization,
   *  automatically created by the JVM.
   */
  boolean isDefault();

  /** An object which the language
   *  runtime deems to be the representation of
a validated
   *  parameter bundle that matches this
specialization.
   *  Typically a list or tuple of reflected
type arguments.
```

```
   *  Always returns {@code null} for default 
specializations.
   *  Specialized fields should depend only on 
this value.
   */
  Object parameters();

  /** An object which the language
   *  runtime deems to be internal data to 
track, and
   *  not a component of the species or type 
parameters.
   *  It might be reflective annotations or 
private behaviors.
   *  Specialized fields should not depend on 
this value,
   *  because their type restrictions are 
resolved when
   *  a species is created.
   */
  Object privateParameter();

  /** The corresponding default specialization 
anchor
   *  for this anchor, which is a very special 
"raw"
   *  specialization.
   *  If this anchor is already the default of 
its kind,
   *  returns {@code this} object.
   */
  SpecializationAnchor defaultSpecialization();

  /** The class whose class file created the
   *  {@code CONSTANT_SpecializationAnchor} 
constant
   *  that created this specialization anchor.
   */
  Class<?> declaringClass();

  /** A value which uniquely distinguishes
   *  {@code CONSTANT_SpecializationAnchor} 
constant
   *  pool entry that created this 
specialization anchor,
   *  within its class file.
   *  An adequate implementation would return 
the
   *  index of the anchor constant in its 
constant pool.
   */
  long specializationAnchorID();

  /** The enclosing specialization anchor, if 
any,
   *  else {@code null}.  Only anchors of
   *  kind {@code PARAM_MethodAndClass} can 
have
   *  enclosing specialization anchors.
   *  An enclosing specialization is always of 
kind
   *  {@code PARAM_Class} and is sub-parametric
   *  to the {@code PARAM_MethodAndClass} 
anchor.
   */
  SpecializationAnchor 
enclosingSpecialization();

  /** The class specialization which this 
anchor exports
   *  else {@code null} if it specializes 
methods only.
   *  Only anchors of kinds {@code PARAM_Class}
   *  and {@code PARAM_MethodAndClass} can 
return
   *  non-null.
   */
  Species species();

  /** Reflection of parametric information.
   *  The lookup object is checked against the 
declaring
   *  class to unlock access to this 
information.
   *  The format of these lists is TBD.
   */
  List<Object> parametricSuperList(Lookup 
lookup);
  List<Object> parametricFieldList(Lookup 
lookup);
  List<Object> parametricMethodList(Lookup 
lookup);
}

public sealed
interface SpecializationAnchorBuilder
  permits SpecializationAnchorBuilder.Impl
{
  /** A pointer to the anchor being built, in a 
mutable
   *  larval form.  The JVM cannot use it until 
the
   *  builder builds the adult form.
   */
  SpecializationAnchor larva();

  /** Initialize the validated parameter bundle 
of the
   *  anchor.  This must be done exactly once,
   *  and before the larva is promoted to 
adult.
   *  The value must not be null.
   *  The value is immediately observable in 
the
   *  larval anchor object.
   */
  void setupParameters(Object obj);

  /** Initialize the private parameter of the
   *  anchor.  This may be done at most once,
   *  and before the larva is promoted to 
adult.
   *  The value must not be null.
   *  The value is immediately observable in 
the
   *  larval anchor object.
   */
  void setupPrivateParameter(Object obj);

  /** Associate this new specialization with a 
pre-existing
   *  species from a previous specialization.
   *  The anchor of the pre-existing species 
must match
   *  this specialization anchor (both class 
and anchor ID).
```

```
   *  This creates a one-to-many relation 
between a single
   *  species and multiple specializations.
   *  The anchor must be of kind {@code 
PARAM_Class},
   *  and must not already have a species set.
   *  The set species is immediately observable 
in the
   *  larval anchor object.
   *  <p>
   *  If this method is not called,
   *  the JVM will create a fresh species 
automatically,
   *  when the larva is promoted to adult, or
   *  when the larva is queried for its 
species,
   *  whichever comes first.
   *  After that, any call to setupSpecies is 
invalid.
   */
  void setupSpecies(Species species);

  /** Build the adult form of the anchor.
   *  At this point any required species
   *  is built unless it has already been set.
   *  A validated parameter bundle must already 
be set.
   *  A private parameter may or may not be 
set,
   *  and if not set will be reported as null.
   */
  SpecializationAnchor build();

  /** Start building a new specialization 
anchor,
   *  starting with a pre-existing one as a 
template.
   *  The template must be a default 
specialization
   *  anchor.
   *  The lookup must be a private-access 
lookup for
   *  the class declaring the specialization.
   */
  static SpecializationAnchorBuilder
    start(Lookup lookup, SpecializationAnchor 
template);
}
```
(More TBW, but see comments scattered throughout.)

Appendix: Translation Tricks and Strategems

Optional fields

Sometimes specialized types need optional fields. For example, a numeric type which supports `NaN` might need an extra boolean field if its underlying type doesn't already have a `NaN` encoding.

If a field is type-restricted to a zero-bit primitive type then it (presumably) occupies no space in its container. Loading this field produces a constant value (the only value of that zero-bit type). Storing a value into such a field must first cast to the zero-bit type, which requires that the value being stored is just another copy of the singleton value that populates the type. Anything other value stored (a null or a different species) will elicit an exception.

Such a field can be viewed as an optional field which has been discarded (for a particular species of the container).

In addition, we may define some sentinel *empty type* which has no values at all, not even a default. Such a paradoxical type, if assigned as a type restriction on a field, would ensure that any access to the field must elicit an exception. This would be a stronger version of an optional field, one unwilling even to produce the default singleton value of some zero-bit type.

Such a field would be regarded (by the user and the JVM) as somehow present but "poisoned"; if you touch it you throw an exception.

Optional methods

If a method can be type-restricted so radically that it cannot be invoked, then the result is as if the species has omitted that method. This is sometimes useful, as when a box type supports comparison, but only if its contained type supports comparison first. (Or, a sum method makes sense only on a species of stream over things you can add.)

The tricks which allow us to simulate optional fields can be adjusted to simulate optional methods (and constructors). An empty type can be assigned to more or more argument types of the method, ensuring that it cannot be called. Or, an empty type can be assigned to the return value, ensuring that it cannot return.

Better yet, a simple sentinel value (such as `void`) could be accepted by the JVM as a type restriction of *any* method (regardless of argument or return arity), and the JVM would simply refuse to link such calls. Such a method would be regarded (by the user and the JVM) as present but "poisoned".

The ad hoc type restrictions computed by a species on its fields and/or methods could then be used to drive optionality of fields or methods.

Optional super types

Supers are harder to treat as optional with corresponding tricks. This is a topic to investigate further. The problem appears to be that a super `S` of a class `C` must be minimally present on all species of `C`, so that name lookup rules are not disturbed by parametric effects.

A partial solution would be to treat all methods of some optional interface as optional, in the sense outlined just above. But this appears to be invasive, because the optional interface might quite innocently wish to have invariant methods; these would not be subject to type restrictions (unless we make new restriction mechanisms besides specialization).

Another possible solution would be to allow some sort of "empty" specialization for an optional super that effectively nullifies all inheritance from that super. (Reflective or "raw" accesses would still reach the super, of course, but perhaps type restrictions would dynamically block all calls to methods on the super.)

A more promising solution is to allow an ad hoc type restriction of the super type to some sentinel (`void` again?) for a specialized super (not the default one), and somehow break field and method inheritance for such a specialization, so that methods and fields resolved by inheritance from the blocked super are themselves "poisoned" by the same mechanism as for piecemeal optional methods.

Optionality of object identity

In Valhalla, the association of object identity with types is flexible. A type that is a primitive class has no instances with object identity. All instances of a type that is an identity class have object identity. In between those extremes, some types (`Object`, interfaces, perhaps some abstract classes) allow a mix of subtypes, identity classes, primitive classes, or both.

Specialization can extend this flexibility to the level of the species type hierarchy, as follows. First, the class as a whole is defined as an abstract which allows both identity and primitive classes. (This would follow rules yet to be finalized, but perhaps the class as a whole is merely an interface, endowed with static factory methods.) Second, the class is made parametric, with a bootstrap method that selects a "type kind" (primitive, identity, or abstract) based on the proposed linkage parameter value. Third, the JVM supplies a species construction factory that allows the "type kind" to be determined in a way that is decoupled from the "type kind" of the variant class.

This might not work, or might require special pleading, if the "type kind" of a type is rigorously defined in terms of the class file supers, and not (also) on some special flag bit (e.g., `ACC_PRIMITIVE`). It seems not impossible that a species factory could start with a variant interface and come up with either kind of concrete implementation.

If all this were possible, then the class `java.lang.Integer` could be retrofitted to support both old-school identity instances and new primitive instances, by manipulating its parametric variance.

Appendix: False Starts and Roads Not Taken

In some interesting cases the JVM is already able to recognize, today, that a dynamic value or type is actually a static constant. If this happens, the JIT can "fold" it into optimized code. After subsequent devirtualizations and inlinings, the resulting code can avoid lots of virtual dispatch and boxing, and boil down hot loops to their essential operations.

Here are some of those cases:

- *inlined call chain:* A caller uses an `ldc` or `static final` to send a static (or statically typed) value *X* as a dynamic argument down a call chain. If the whole call chain is inlined into a single JIT compilation task, then constant propagation turns the dynamic value into a static value, or at least gives it a static type. Precondition: Inlining the callee into the caller.

- *type speculation:* A caller chooses a static type *T* and passes an object *A* of that type under a dull type (like `Object`). The callee guesses that *A* is of type *T*, and after verifying that is the case, can use *T* as a static type. Precondition: The callee should be able to guess all relevant types *T* from all callers; the practical maximum number is 2 or 3 distinct types.

- *cast to type:* A caller chooses a static type *T* and passes an object *A* of that type under a dull type (like `Object`). If the callee can be induced to treat the value `T.class` as a constant, then it can compute `T.class.cast(A)` (or the equivalent `instanceof` instruction) and can then access all of the *T*-features of *A*. Precondition: Positioning the dynamic type *T* as a static value in the callee.

- *trusted final:* A caller stores a static constant *X* into a holder object *Y*, using a trusted final field. If *Y* can be treated as a static value, so can *X*. Precondition: Positioning the holder object *Y* as a static value in the callee.

- *customized method handle:* A method handle M_0; is bound to an argument value *X*, yielding a new method handle M_1. If M_1; is subsequently recompiled, then *X* becomes a static value within the compilation of M_1. Precondition: Calling the hot path through M_1.

- *the constant pool*: A caller decides some global static value is needed, and arranges to store it in a constant pool. The value can be something built in or an arbitrary value (using `invokedynamic` or `CONSTANT_Dynamic`). Precondition: The number of cached values must be fixed at class load time, and caller and callee must somehow share access to a constant pool holding the values.

Note that these cases all depend on specialized preconditions. Some are under control of the programmer, while others depend on JVM heuristics.

One simple example where none of the above tactics help is a B-tree library where all the arrays are of a common length (say, 64) but the JVM is forced to check at every array reference that the array length is, once again, 64. The value 64 is surely declared prominently somewhere as a static constant, and yet by the time it is stored in the header of an array, it has become indistinguishable from a dynamic value. Yes, we could add an optimization for array-length profiling and speculation; maybe we will someday, but there are many similar problems of the same sort. Covering them all seems to be an unending game of whack-a-mole. What's needed is help from the user to keep static values and types static, even in places where, today, they "decay" into dynamic values and types after parameter passing.

The root difficulty with passing a static value or type as a dynamic parameter is that its static character is obscured. Callers and callees are often decoupled and processed by different JIT tasks, especially when a callee is a reusable algorithm. After decoupling, a constant in a caller becomes difficult to recover as a constant in a callee, even by speculative or heroic optimizations.

The new parametric constants proposed here overcome that root difficulty exactly when inlining fails: A static decision about a linkage parameter is bound into a caller, and becomes available in the callee, *even when the callee fails to be inlined*. The specialization decision is recorded in the caller, and is perhaps shared among multiple callers, as with class layouts. The JVM is then given the option to customize multiple versions of the callee, based on the behavior (especially the "hotness") of the various callers.

Non-proposals

What about templates in the static compiler?
C++ has a code customization mechanism called templates. They allow a wide variety of arguments, including types, primitive values, and functions. Within the context of a template, the template arguments are reliably treated as constants. The arguments to a template are thus true static type (and value and function) parameters. There is a big downside: C++ templates are resolved and fully compiled before the program executes. This workflow does not fit well into Java's paradigm of dynamic class loading and lazy linking and initialization. In addition, any mechanism that eagerly generates many customized versions of the same bytecode will tend to load down the class loader and JIT. A better fit would be a mechanism which would allow expansion during the JVM's dynamic link phases, or even later, when the JIT optimizes hot code paths.

What about more and better inlining?
It is quite true that many programs can exhibit optimized behavior equivalent to customization of data and code, given enough rounds of the following optimizations in the JIT:

- inline a callee into a caller
- propagate constants from the caller to the inlined callee
- deduce types of data shared by caller and callee
- customize the inlined caller code using the value types and constants
- lift shared data structures out of the heap
- customize those data structures to the value types and constants actually used
- when information is missing, try profiling and speculating

This is a very powerful toolkit of techniques which collectively make Java competitive with languages that are statically compiled and linked. We can and will ask the JIT to work harder on specialized generics, but there are three limitations to the above toolkit which are exacerbated by specialization:

1. Inlining is not reliable. Deep call chains *must* include out of line calls. Also, generic code, because of its greater reusability, may be factored into relatively deeper call chains.

2. Heap structures can only be transformed after the JIT runs. Data created during JVM warmup cannot assume JIT optimizations, and yet must support full speed processing, if it survives.

3. Speculation becomes less accurate as types and values become more differentiated due to type specialization, and/or profiles become more polluted due to sharing of well-factored generic code.

Explicit specialization signals, captured at the JVM level, at link resolution time, between caller and callee, provide the framework the JVM needs to produce customized data structures up front, and customized code as hot spots develop, even where inlining fails (as it sometimes must).

Weren't you implementing specialization via bytecode spinning?
Earlier prototypes of Valhalla specialization used bytecode spinning, so that each specialization of a class or interface had its own class file, with bytecodes (and other information) customized to the required types. Class loaders could be "hooked" to spin specialized versions of a type on demand.

This was a good way to experiment with (some) language designs and flesh out requirements, but it didn't hang together well enough to continue with. Here is a partial list of reasons the approach didn't pan out:

- Subclassing (with overrides) and specialization are independent dimensions of type variation, so implementing them using the same mechanism causes conflicts.

- Wildcard and raw types don't have a natural relation to other specializations of the same type, when overrides are use to model the interconnections.

- Specialization at class load time commits the JVM to a separate copy of specialized code for each specialization. In essence, there is no separate, later choice to customize specializations based on profile feedback (as may be done in the Parametric VM). The JIT has to separately compile and optimize load-time specializations whether or not the extra work is profitable.

- However much (or little) the trick works for specialized *types*, spinning instances of specialized *methods* seems to require lifting each generic method into its own class, which is a large overhead.

- Specialized fields must be wrapped in access methods, and each specialized data structure must be represented as a new class. Such classes "leak" into the user model as classes that the user didn't intend to create.

- Generation of specialized bytecodes from a pre-existing template is a complicated business. One corner case gives a flavor of the kind of problem that arises: If a type variable is replaced by `long`, suddenly the stack effects of affected internal variables must be expanded to stack slot pairs, with relevant bytecode changes. In general, there is little assurance that specialized bytecodes can be generated from some intermediate form, short of recompiling the source code for each specialization.

What about dependent types in the VM?
Dependent types are a theoretical language concept which could address the problem of code and data customization. If the JVM type system were upgraded so that the static types of methods could depend on dynamic values (or types) then users could choose to route static specialization information through a shim of dependent types. And the JVM would surely do the right thing. There are two problems: Such type systems are poorly understood, and their connections to the existing optimization tactics of the JVM are even less understood. In any case, this would be a large change to the JVM type system.

What about extending the language of type descriptors?
One light version of dependent types, in the JVM, would be a way of introducing descriptors (of fields, arguments, and returns) which include "holes" filled by resolved type information, differently at different points in the program. This could be specified and engineered, at the cost of reinventing the JVM's symbolic resolution mechanisms and type systems, to extend the syntax and semantics of descriptors and class names, to take account of such "holes". But it is much easier to plumb such dependencies through a separate channel (as in this proposal), which leaves descriptors untouched. The dependencies are similar, but the paths by which they are introduced are through a cleanly factored side channel, not a complexification of the JVM's type and descriptor system. One benefit of such a factoring is that, in the setting of such a side channel, language-level semantics can be more readily defined by reference library code invoked by a bootstrap method, not descriptor processing logic hardwired into the JVM.

In the end, it seems likely that whatever might be done with enhanced descriptors could also be represented with parametric side channels (assuming they are "just as constant" as the descriptors being represented). The side channel approach is simpler and cleaner to engineer in the JVM.

In some very narrow cases, enhanced type descriptors might possibly assist in organizing method overloads, such as `m(List<InlineDouble>)` versus `m(List<InlineInt>)`. The JVM could possibly assist with this by allowing the parameter tokens (`<InlineDouble>` and `<InlineInt>`) be stored in a side-channel associated with the symbolic reference, perhaps a name mangling or some other place. For this to work, the JVM would not be required to interpret those extra descriptor tokens. Alternatively, those extra tokens could be used to derive implicit type restrictions to apply to the affected methods. All of this is doable, but none of it has very compelling use cases. What is interesting, though, is it seems possible to layer traditional (CLR-style) parametric type descriptor syntaxes on top of this parametric VM design, as sugar that expands into the lower-level primitives of this proposal.

What about type tokens in `this`?
On paper, the problem of representing type variables of all kinds can be reduced to representing species information in ad hoc object fields, secretly injected by javac or by the JVM.

This is option 1(a) or 1(b) as discussed in section 4.2 of Kennedy and Syme's CLR generics paper. This may also be a good way to prototype the plumbing of specialization information. However, adding even one extra word to every specialized object would be a noticeable overhead, especially for small objects. Inevitably, it turns out to be preferable to access specialization information via the pre-existing runtime type pointer at the head of every Java object. This is the route taken by us and by CLR.

Noticing that all API points (not just generic classes) benefit from parametricity, we could try to pass type tokens through the bytecodes which manage method and even field access, as well as class creation. Pushing on this goal, we find that constant pool slots already carry resolved symbolic references, and thus are an ideal place to store whatever additional data works like type tokens. Turning to the problem of managing that data at the definition site of parametric API points, we find that we need constant pool structures to work with specialization anchors, which become the source of type tokens, if those are used. The constant pool structures are the primitives, and the type tokens are translation artifacts that can be plumbed as needed.

Method specialization information could also be encoded using invisible synthetic helper instances, created on each call. Also, each generic method might be relocated, from the class it is declared in, to a synthetic inner helper class (perhaps one per generic method). Normal nested class links would allow the method to access the real `this` as well as the synthetic helper. Such an approach is disruptive to translation strategy, creating many synthetic classes "under the hoods", which the JVM has to untangle in order to optimize. The helper instances would look like regular objects to the JVM, and so would not provide clearly marked points for the JVM to invest customization effort, compared to a purpose-built specialization framework in the classfile. These simulation overheads would not be present in a corner case, but rather would appear wherever a factory method serves to create generic instances: Clearly, a factory method is *not* able to refer to type variable bindings encoded in an instance, since its job is to create the instance. The simulation in such a case requires the caller to first build a helper object to contain the type variable bindings, and then immediately copy those bindings into the real, user-visible object.

Given that the JVM *must* have special data paths to manage customizable layouts of objects created by the `new` instruction inside the factory method, it is a no-brainer to ditch the helper object and plumb those special data paths (with the help of a split constant pool) all the way out through the factory method and to its callers. The use case of factory methods is one reason the emphasis in the present design is on uniform specialization of *all* API points, not a type-only specialization mechanism.

(Extending uniform specialization to fields as well as types and methods provides an apt way to represent and process variant fields in customizable layouts, and simpler options for extensions in the future, such as species statics.)