orphan:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\ABI\(swift-main)(docs)(ABI)TypeMetadata.rst, line 5)

Unknown directive type "highlight".

```
.. highlight:: none
```

Type Metadata

The Swift runtime keeps a **metadata record** for every type used in a program, including every instantiation of generic types. These metadata records can be used by (TODO: reflection and) debugger tools to discover information about types. For non-generic nominal types, these metadata records are generated statically by the compiler. For instances of generic types, and for intrinsic types such as tuples, functions, protocol compositions, etc., metadata records are lazily created by the runtime as required. Every type has a unique metadata record; two **metadata pointer** values are equal iff the types are equivalent.

In the layout descriptions below, offsets are given relative to the metadata pointer as an index into an array of pointers. On a 32-bit platform, **offset 1** means an offset of 4 bytes, and on 64-bit platforms, it means an offset of 8 bytes.

Common Metadata Layout

All metadata records share a common header, with the following fields:

- The **value witness table** pointer references a vtable of functions that implement the value semantics of the type, providing fundamental operations such as allocating, copying, and destroying values of the type. The value witness table also records the size, alignment, stride, and other fundamental properties of the type. The value witness table pointer is at **offset -1** from the metadata pointer, that is, the pointer-sized word **immediately before** the pointer's referenced address.
- The **kind** field is a pointer-sized integer that describes the kind of type the metadata describes. This field is at **offset 0** from the metadata pointer.

  The current kind values are as follows:

  - Class metadata has of kind of **0** unless the class is required to interoperate with Objective-C. If the class is required to interoperate with Objective-C, the kind field is instead an *isa pointer* to an Objective-C metaclass. Such a pointer can be distinguished from an enumerated metadata kind because it is guaranteed to have a value larger than **2047**. Note that this is a more basic sense of interoperation than is meant by the `@objc` attribute: it is what is required to support Objective-C message sends and retain/release. All classes are required to interoperate with Objective-C on this level when building for an Apple platform.
  - Struct metadata has a kind of **1**.
  - Enum metadata has a kind of **2**.
  - Optional metadata has a kind of **3**.
  - **Opaque metadata** has a kind of **8**. This is used for compiler `Builtin` primitives that have no additional runtime information.
  - Tuple metadata has a kind of **9**.
  - Function metadata has a kind of **10**.
  - Protocol metadata has a kind of **12**. This is used for protocol types, for protocol compositions, and for the `Any` type.
  - Metatype metadata has a kind of **13**.
  - Objective C class wrapper metadata has a kind of **14**.
  - Existential metatype metadata has a kind of **15**.

Struct Metadata

In addition to the common metadata layout fields, struct metadata records contain the following fields:

- The nominal type descriptor is referenced at **offset 1**.
- If the struct is generic, then the generic argument vector begins at **offset 2**.
- A vector of **field offsets** begins immediately after the generic argument vector. For each field of the struct, in `var` declaration order, the field's offset in bytes from the beginning of the struct is stored as a pointer-sized integer.

Enum Metadata

In addition to the common metadata layout fields, enum metadata records contain the following fields:

- The nominal type descriptor is referenced at **offset 1**.
- If the enum is generic, then the generic argument vector begins at **offset 2**.

Optional Metadata

Optional metadata share the same basic layout as enum metadata. They are distinguished from enum metadata because of the importance of the `Optional` type for various reflection and dynamic-casting purposes.

Tuple Metadata

In addition to the common metadata layout fields, tuple metadata records contain the following fields:

- The **number of elements** in the tuple is a pointer-sized integer at **offset 1**.
- The **labels string** is a pointer to the labels for the tuple elements at **offset 2**. Labels are encoded in UTF-8 and separated by spaces, and the entire string is terminated with a null character. For example, the labels in the tuple type `(x: Int, Int, z: Int)` would be encoded as the character array `"x  z \0"`. A label (possibly zero-length) is provided for each element of the tuple, meaning that the label string for a tuple of **n** elements always contains exactly **n** spaces. If the tuple has no labels at all, the label string is a null pointer.
- The **element vector** begins at **offset 3** and consists of an array of type-offset pairs. The metadata for the *n*th element's type is a pointer at **offset 3+2*n**. The offset in bytes from the beginning of the tuple to the beginning of the *n*th element is at **offset 3+2*n+1**.

Function Metadata

In addition to the common metadata layout fields, function metadata records contain the following fields:

- The function flags are stored at **offset 1**. This includes information such as the semantic convention of the function, whether the function throws, and how many parameters the function has.
- A reference to the **result type** metadata record is stored at *offset 2**. If the function has multiple returns, this references a tuple metadata record.
- The **parameter type vector** follows the result type and consists of **NumParameters** type metadata pointers corresponding to the types of the parameters.
- The optional **parameter flags vector** begins after the end of **parameter type vector** and consists of **NumParameters** 32-bit flags. This includes information such as whether the parameter is `inout` or whether it is variadic. The presence of this vector is signalled by a flag in the function flags; if no parameter has any non-default flags, the flag is not set.

Currently we have specialized ABI endpoints to retrieve metadata for functions with 0/1/2/3 parameters - *swift_getFunctionTypeMetadata{0|1|2|3}* and the general one *swift_getFunctionTypeMetadata* which handles all other function types and functions with parameter flags e.g. *(inout Int) -> Void*. Based on the usage information collected from Swift Standard Library and Overlays as well as Source Compatibility Suite it was decided not to have specialized ABI endpoints for functions with parameter flags due their minimal use.

Protocol Metadata

In addition to the common metadata layout fields, protocol metadata records contain the following fields:

- A **layout flags** word is stored at **offset 1**. The bits of this word describe the existential container layout used to represent values of the type. The word is laid out as follows:
  - The **number of witness tables** is stored in the least significant 24 bits. Values of the protocol type contain this number of witness table pointers in their layout.
  - The **special protocol kind** is stored in 6 bits starting at bit 24. Only one special protocol kind is defined: the *Error* protocol has value 1.
  - The **superclass constraint indicator** is stored at bit 30. When set, the protocol type includes a superclass constraint (described below).
  - The **class constraint** is stored at bit 31. This bit is set if the type is **not** class-constrained, meaning that struct, enum, or class values can be stored in the type. If not set, then only class values can be stored in the type, and the type uses a more efficient layout.
- The **number of protocols** that make up the protocol composition is stored at **offset 2**. For the "any" types `Any` or `AnyObject`, this is zero. For a single-protocol type `P`, this is one. For a protocol composition type `P & Q & ...`, this is the number of protocols.
- If the **superclass constraint indicator** is set, type metadata for the superclass follows at the next offset.
- The **protocol vector** follows. This is an inline array of pointers to descriptions of each protocol in the composition. Each pointer references either a Swift protocol descriptor or an Objective-C *Protocol*; the low bit will be set to indicate when it references an Objective-C protocol. For an "any" or "AnyObject" type, there is no protocol descriptor vector.

Metatype Metadata

In addition to the common metadata layout fields, metatype metadata records contain the following fields:

- A reference to the metadata record for the **instance type** that the metatype represents is stored at **offset 1**.

Existential Metatype Metadata

In addition to the common metadata layout fields, existential metatype metadata records contain the following fields:

- A reference to the metadata record for the **instance type** of the metatype is stored at **offset 1**. This is always either an existential type metadata or another existential metatype.
- A word of flags summarizing the existential type are stored at **offset 2**.

Class Metadata

Class metadata is designed to interoperate with Objective-C; all class metadata records are also valid Objective-C `Class` objects. Class metadata pointers are used as the values of class metatypes, so a derived class's metadata record also serves as a valid class metatype value for all of its ancestor classes.

- The **destructor pointer** is stored at **offset -2** from the metadata pointer, behind the value witness table. This function is invoked by Swift's deallocator when the class instance is destroyed.
- The **isa pointer** pointing to the class's Objective-C-compatible metaclass record is stored at **offset 0**, in place of an integer kind discriminator.
- The **super pointer** pointing to the metadata record for the superclass is stored at **offset 1**. If the class is a root class, it is null.
- On platforms which support Objective-C interoperability, two words are reserved for use by the Objective-C runtime at **offset 2** and **offset 3**; on other platforms, nothing is reserved.
- On platforms which support Objective-C interoperability, the **rodata pointer** is stored at **offset 4**; on other platforms, it is not present. The rodata pointer points to an Objective-C compatible rodata record for the class. This pointer value includes a tag. The **low bit is always set to 1** for Swift classes and always set to 0 for Objective-C classes.
- The **class flags** are a 32-bit field at **offset 5** on platforms which support Objective-C interoperability; on other platforms, the field is at **offset 2**.
- The **instance address point** is a 32-bit field following the class flags. A pointer to an instance of this class points this number of bytes after the beginning of the instance.
- The **instance size** is a 32-bit field following the instance address point. This is the number of bytes of storage present in every object of this type.
- The **instance alignment mask** is a 16-bit field following the instance size. This is a set of low bits which must not be set in a pointer to an instance of this class.
- The **runtime-reserved field** is a 16-bit field following the instance alignment mask. The compiler initializes this to zero.
- The **class object size** is a 32-bit field following the runtime-reserved field. This is the total number of bytes of storage in the class metadata object.
- The **class object address point** is a 32-bit field following the class object size. This is the number of bytes of storage in the class metadata object.
- The nominal type descriptor for the most-derived class type is referenced at an offset immediately following the class object address point. On 64-bit and 32-bit platforms which support Objective-C interoperability, this is, respectively, at **offset 8** and at **offset 11**; in platforms that do not support Objective-C interoperability, this is, respectively, at **offset 5** and at **offset 8**.
- For each Swift class in the class's inheritance hierarchy, in order starting from the root class and working down to the most derived class, the following fields are present:
    - First, a reference to the **parent** metadata record is stored. For classes that are members of an enclosing nominal type, this is a reference to the enclosing type's metadata. For top-level classes, this is null.

      TODO: The parent pointer is currently always null.
    - If the class is generic, its generic argument vector is stored inline.
    - The **vtable** is stored inline and contains a function pointer to the implementation of every method of the class in declaration order.
    - If the layout of a class instance is dependent on its generic parameters, then a **field offset vector** is stored inline, containing offsets in bytes from an instance pointer to each field of the class in declaration order. (For classes with fixed layout, the field offsets are accessible statically from global variables, similar to Objective-C ivar offsets.)

  Note that none of these fields are present for Objective-C base classes in the inheritance hierarchy.

Objective C class wrapper metadata

Objective-C class wrapper metadata are used when an Objective-C `Class` object is not a valid Swift type metadata.

In addition to the common metadata layout fields, Objective-C class wrapper metadata records have the following fields:

- A `Class` value at **offset 1** which is known to not be a Swift type metadata.

Generic Argument Vector

Metadata records for instances of generic types contain information about their generic arguments. For each parameter of the type, a reference to the metadata record for the type argument is stored. After all of the type argument metadata references, for each type parameter, if there are protocol requirements on that type parameter, a reference to the witness table for each protocol it is required to conform to is stored in declaration order.

For example, given a generic type with the parameters `<T, U, V>`, its generic parameter record will consist of references to the

metadata records for `T`, `U`, and `V` in succession, as if laid out in a C struct:

```
struct GenericParameterVector {
  TypeMetadata *T, *U, *V;
};
```

If we add protocol requirements to the parameters, for example, `<T: Runcible, U: Fungible & Ansible, V>`, then the type's generic parameter vector contains witness tables for those protocols, as if laid out:

```
struct GenericParameterVector {
  TypeMetadata *T, *U, *V;
  RuncibleWitnessTable *T_Runcible;
  FungibleWitnessTable *U_Fungible;
  AnsibleWitnessTable *U_Ansible;
};
```

Foreign Class Metadata

Foreign class metadata describes "foreign" class types, which support Swift reference counting but are otherwise opaque to the Swift runtime.

- The nominal type descriptor for the most-derived class type is stored at **offset 0**.
- The **super pointer** pointing to the metadata record for the superclass is stored at **offset 1**. If the class is a root class, it is null.
- Three **pointer-sized fields**, starting at **offset 2**, are reserved for future use.

Nominal Type Descriptor

Warning: this is all out of date!

The metadata records for class, struct, and enum types contain a pointer to a **nominal type descriptor**, which contains basic information about the nominal type such as its name, members, and metadata layout. For a generic type, one nominal type descriptor is shared for all instantiations of the type. The layout is as follows:

- The **kind** of type is stored at **offset 0**, which is as follows:
  - **0** for a class,
  - **1** for a struct, or
  - **2** for an enum.
- The mangled **name** is referenced as a null-terminated C string at **offset 1**. This name includes no bound generic parameters.
- The following four fields depend on the kind of nominal type.
  - For a struct or class:
    - The **number of fields** is stored at **offset 2**. This is the length of the field offset vector in the metadata record, if any.
    - The **offset to the field offset vector** is stored at **offset 3**. This is the offset in pointer-sized words of the field offset vector for the type in the metadata record. If no field offset vector is stored in the metadata record, this is zero.
    - The **field names** are referenced as a doubly-null-terminated list of C strings at **offset 4**. The order of names corresponds to the order of fields in the field offset vector.
    - The **field type accessor** is a function pointer at **offset 5**. If non-null, the function takes a pointer to an instance of type metadata for the nominal type, and returns a pointer to an array of type metadata references for the types of the fields of that instance. The order matches that of the field offset vector and field name list.
  - For an enum:
    - The **number of payload cases** and **payload size offset** are stored at **offset 2**. The least significant 24 bits are the number of payload cases, and the most significant 8 bits are the offset of the payload size in the type metadata, if present.
    - The **number of no-payload cases** is stored at **offset 3**.
    - The **case names** are referenced as a doubly-null-terminated list of C strings at **offset 4**. The names are ordered such that payload cases come first, followed by no-payload cases. Within each half of the list, the order of names corresponds to the order of cases in the enum declaration.
    - The **case type accessor** is a function pointer at **offset 5**. If non-null, the function takes a pointer to an instance of type metadata for the enum, and returns a pointer to an array of type metadata references for the types of the cases of that instance. The order matches that of the case name list. This function is similar to the field type accessor for a struct, except also the least significant bit of each element in the result is set if the enum case is an **indirect case**.
- If the nominal type is generic, a pointer to the **metadata pattern** that is used to form instances of the type is stored at **offset 6**. The pointer is null if the type is not generic.
- The **generic parameter descriptor** begins at **offset 7**. This describes the layout of the generic parameter vector in the metadata record:
  - The **offset of the generic parameter vector** is stored at **offset 7**. This is the offset in pointer-sized words of the generic parameter vector inside the metadata record. If the type is not generic, this is zero.
  - The **number of type parameters** is stored at **offset 8**. This count includes associated types of type parameters with protocol constraints.

    - The **number of type parameters** is stored at **offset 9**. This count includes only the primary formal type parameters.
    - For each type parameter **n**, the following fields are stored:
        - The **number of witnesses** for the type parameter is stored at **offset 10+n**. This is the number of witness table pointers that are stored for the type parameter in the generic parameter vector.

Note that there is no nominal type descriptor for protocols or protocol types. See the protocol descriptor description below.

Protocol Descriptor

Protocol descriptors describe the requirements of a protocol, and act as a handle for the protocol itself. They are referenced by Protocol metadata, as well as Protocol Conformance Records and generic requirements. Protocol descriptors are only created for non-*@objc* Swift protocols: *@objc* protocols are emitted as Objective-C metadata. The layout of Swift protocol descriptors is as follows:

- Protocol descriptors are context descriptors, so they are prefixed by context descriptor metadata. (FIXME: these are not yet documented)
- The 16-bit kind-specific flags of a protocol are defined as follows:
    - **Bit 0** is the **class constraint bit**. It is set if the protocol is **not** class-constrained, meaning that any struct, enum, or class type may conform to the protocol. It is unset if only classes can conform to the protocol.
    - **Bit 1** indicates that the protocol is **resilient**.
    - **Bits 2-7** indicate specify the **special protocol kind**. Only one special protocol kind is defined: the *Error* protocol has value 1.
- A pointer to the **name** of the protocol.
- The number of generic requirements within the **requirement signature** of the protocol. The generic requirements themselves follow the fixed part of the protocol descriptor.
- The number of **protocol requirements** in the protocol. The protocol requirements follow the generic requirements that form the **requirement signature**.
- A string containing the **associated type names**, a C string comprising the names of all of the associated types in this protocol, separated by spaces, and in the same order as they appear in the protocol requirements.
- The **generic requirements** that form the **requirement signature**.
- The **protocol requirements** of the protocol.

Protocol Conformance Records

A *protocol conformance record* states that a given type conforms to a particular protocol. Protocol conformance records are emitted into their own section, which is scanned by the Swift runtime when needed (e.g., in response to a *swift_conformsToProtocol()* query). Each protocol conformance record contains:

- The protocol descriptor describing the protocol of the conformance, represented as an (possibly indirect) 32-bit offset relative to the field. The low bit indicates whether it is an indirect offset; the second lowest bit is reserved for future use.
- A reference to the **conforming type**, represented as a 32-bit offset relative to the field. The lower two bits indicate how the conforming type is represented:

    0. A direct reference to a nominal type descriptor.
    1. An indirect reference to a nominal type descriptor.
    2. Reserved for future use.
    3. A reference to a pointer to an Objective-C class object.

- The **witness table field** that provides access to the witness table describing the conformance itself, represented as a direct 32-bit relative offset. The lower two bits indicate how the witness table is represented:

    0. The **witness table field** is a reference to a witness table.
    1. The **witness table field** is a reference to a **witness table accessor** function for an unconditional conformance.
    2. The **witness table field** is a reference to a **witness table accessor** function for a conditional conformance.
    3. Reserved for future use.

- A 32-bit value reserved for future use.

Recursive Type Metadata Dependencies

The Swift type system is built up inductively by the application of higher-kinded type constructors (such as "tuple" or "function", as well as user-defined generic types) to other, existing types. Crucially, it is the "least fixed point" of that inductive system, meaning that it does not include **infinite types** (Âµ-types) whose basic identity can only be defined in terms of themselves.

That is, it is possible to write the type:

```
typealias IntDict = Dictionary<String, Int>
```

but it is not possible to directly express the type:

```
typealias RecursiveDict = Dictionary<String, RecursiveDict>
```

However, Swift does permit the expression of types that have recursive dependencies upon themselves in ways other than their basic identity. For example, class `A` may inherit from a superclass `Base<A>`, or it may contain a field of type `(A, A)`. In order to support the dynamic reification of such types into type metadata, as well as to support the dynamic layout of such types, Swift's metadata runtime supports a system of metadata dependency and iterative initialization.

Metadata States

A type metadata may be in one of several different dynamic states:

- An **abstract** metadata stores just enough information to allow the identity of the type to be recovered: namely, the metadata's kind (e.g. **struct**) and any kind-specific identity information it entails (e.g. the nominal type descriptor and any generic arguments).
- A **layout-complete** metadata additionally stores the components of the type's "external layout", necessary to compute the layout of any type that directly stores a value of the type. In particular, a metadata in this state has a meaningful value witness table.
- A **non-transitively complete** metadata has undergone any additional initialization that is required in order to support basic operations on the type. For example, a metadata in this state will have undergone any necessary "internal layout" that might be required in order to create values of the type but not to allocate storage to hold them. For example, a class metadata will have an instance layout, which is not required in order to compute the external layout, but is required in order to allocate instances or create a subclass.
- A **complete** metadata additionally makes certain guarantees of transitive completeness of the metadata referenced from the metadata. For example, a complete metadata for `Array<T>` guarantees that the metadata for `T` stored in the generic arguments vector is also complete.

Metadata never backtrack in their state. In particular, once metadata is complete, it remains complete forever.

Transitive Completeness Guarantees

A complete class metadata makes the following guarantees:

- Its superclass metadata (if it has a superclass) is complete.
- Its generic arguments (if it has any) are complete.
- By implication, the generic arguments of its superclasses are complete.

A complete struct, enum, or optional metadata makes the following guarantees:

- Its generic arguments (if it has any) are complete.

A complete tuple metadata makes the following guarantees:

- Its element types are complete.

Other kinds of type metadata do not make any completeness guarantees. The metadata kinds with transitive guarantees are the metadata kinds that potentially require two-phase initialization anyway. Other kinds of metadata could otherwise declare themselves complete immediately on allocation, so the transitive completeness guarantee would add significant complexity to both the runtime interface and its implementation, as well as adding probably-unrecoverable memory overhead to the allocation process.

It is also true that it is far more important to be able to efficiently recover complete metadata from the stored arguments of a generic type than it is to be able to recover such metadata from a function metadata.

Completeness Requirements

Type metadata are required to be transitively complete when they are presented to most code. This allows that code to work with the metadata without explicitly checking for its completeness. Metadata in the other states are typically encountered only when initializing or building up metadata.

Specifically, a type metadata record is required to be complete when:

- It is passed as a generic argument to a function (other than a metadata access function, witness table access function, or metadata initialization function).
- It is used as a metatype value, including as the `Self` argument to a `static` or `class` method, including initializers.
- It is used to build an opaque existential value.

Metadata Requests and Responses

When calling a metadata access function, code must provide the following information:

- the required state of the metadata, and
- whether the callee should block until the metadata is available in that state.

The access function will then return:

- the metadata and
- the current dynamic state of the metadata.

Access functions will always return the correct metadata record; they will never return a null pointer. If the metadata has not been allocated at the time of the request, it will at least be allocated before the access function returns. The runtime will block the current thread until the allocation completes if necessary, and there is currently no way to avoid this.

Since access functions always return metadata that is at least in the abstract state, it is not meaningful to make a non-blocking request for abstract metadata.

The returned dynamic state of the metadata may be less than the requested state if the request was non-blocking. It is not otherwise affected by the request; it is the known dynamic state of the metadata at the time of the call. Note that of course this dynamic state is just a lower bound on the actual dynamic state of the metadata, since the actual dynamic state may be getting concurrently advanced by another thread.

In general, most code should request metadata in the **complete** state (as discussed above) and should block until the metadata is available in that state. However:

- When requesting metadata solely to serve as a generic argument of another metadata, code should request **abstract** metadata. This can potentially unblock cycles involving the two metadata.
- Metadata initialization code should generally make non-blocking requests; see the next section.

Metadata access functions that cache their results should only cache if the dynamic state is complete; this substantially simplifies the caching logic, and in practice most metadata will be dynamically complete. Note that this rule can be applied without considering the request.

Code outside of the runtime should never attempt to ascertain a metadata's current state by inspecting it, e.g. to see if it has a value witness table. Metadata initialization is not required to use synchronization when initializing the metadata record; the necessary synchronization is done at a higher level in the structures which record the metadata's dynamic state. Because of this, code inspecting aspects of the metadata that have not been guaranteed by the returned dynamic state may observe partially-initialized state, such as a value witness table with a meaningless size value. Instead, that code should call the `swift_checkMetadataState` function.

Metadata Allocation and Initialization

In order to support recursive dependencies between type metadata, the creation of type metadata is divided into two phases:

- allocation, which creates an abstract metadata, and
- initialization, which advances the metadata through the progression of states.

Allocation cannot fail. It should return relatively quickly and should not make any metadata requests.

The initialization phase will be repeatedly executed until it reaches completion. It is only executed by one thread at a time. Compiler-emitted initialization functions are given a certain amount of scratch space that is passed to all executions; this can be used to skip expensive or unrepeatable steps in later re-executions.

Any particular execution of the initialization phase can fail due to an unsatisfied dependency. It does so by returning a **metadata dependency**, which is a pair of a metadata and a required state for that metadata. The initialization phase is expected to make only non-blocking requests for metadata. If a response does not satisfy the requirement, the returned metadata and the requirement should be presented to the caller as a dependency. The runtime does two things with this dependency:

- It attempts to add the initialization to the **completion queue** of the dependent metadata. If this succeeds, the initialization is considered blocked; it will be unblocked as soon as the dependent metadata reaches the required state. But it can also fail if the dependency is already resolved due to concurrent initialization; if so, the initialization is immediately resumed.
- If it succeeds in blocking the initialization on the dependency, it will check for an unresolvable dependency cycle. If a cycle exists, it will be reported on stderr and the runtime will abort the process. This depends on the proper use of non-blocking requests; the runtime does not make any effort to detect deadlock due to cycles of blocking requests.

Initialization must not repeatedly report failure based on stale information about the dynamic state of a metadata. (For example, it must not cache metadata states from previous executions in the initialization scratch space.) If this happens, the runtime may spin, repeatedly executing the initialization phase only to have it fail in the same place due to the same stale dependency.

Compiler-emitted initialization functions are only responsible for ensuring that the metadata is **non-transitively complete**. They signal this by returning a null dependency to the runtime. The runtime will then ensure transitive completion. The initialization function should not try to "help out" by requesting complete metadata instead of non-transitively-complete metadata; it is impossible to resolve certain recursive transitive-closure problems without the more holistic information available to the runtime. In general, if an initialization function seems to require transitively-complete metadata for something, try to make it not.

If a compiler-emitted initialization function returns a dependency, the current state of the metadata (**abstract** vs. **layout-complete**) will be determined by inspecting the **incomplete** bit in the flags of the value witness table. Compiler-emitted initialization functions are therefore responsible for ensuring that this bit is set correctly.