Dynamic Casting Behavior

- Author: Tim Kientzle
- Implementation: apple/swift#29658

Introduction

The Swift language has three casting operators: `is` , `as?` , and `as!` . Each one takes an instance on the left-hand side and a type expression on the right-hand side.

- The *cast test operator* `is` tests whether a particular instance can be converted to a particular destination type. It returns a boolean result.
- The *conditional cast operator* `as?` attempts the conversion and returns an `Optional` result: `nil` if the conversion failed, and a non-nil optional containing the result otherwise.
- The *forced cast operator* `as!` unconditionally performs the casting conversion and returns the result. If an `as!` expression does not succeed, the implementation may terminate the program.

Note: The static coercion operator `as` serves a different role and its behavior will not be specifically discussed in this document.

The following invariants relate the three casting operators:

- Cast test: `x is T == ((x as? T) != nil)`
- Conditional cast: `(x as? T) == (x is T) ? .some(x as! T) : nil`
- Forced cast: `x as! T` is equivalent to `(x as? T)!`

In particular, note that `is` and `as!` can be implemented in terms of `as?` and vice-versa.

As with other operators with `!` in their names, `as!` is intended to only be invoked in cases where the programmer knows a priori that the cast will succeed. If the conversion would not succeed, then the behavior may not be fully deterministic. See the discussion of Array casting below for a specific case where this non-determinism can be important.

The following sections detail the rules that govern the casting operations for particular Swift types. Except where noted, casting between types described in different sections below will always fail -- for example, casting a struct instance to a function type or vice-versa.

Where possible, each section includes machine-verifiable *invariants* that can be used as the basis for developing a robust test suite for this functionality.

Identity Cast

Casting an instance of a type to its own type will always succeed and return the original value unchanged.

```
let a: Int = 7
a is Int // true
a as? Int // Succeeds
a as! Int == a // true
```

Class and Foreign Types

Class types generally follow standard object-oriented casting conventions. Objective-C and CoreFoundation (CF) types follow the behaviors expected from Objective-C.

Classes

Casting among class types follows standard object-oriented programming conventions:

- Class upcasts: If `C` is a subclass of `SuperC` and `c` is an instance of `C`, then `c is SuperC == true` and `(c as? SuperC) != nil`. (Note: These "upcasts" do not change the in-memory representation.) However, when `c` is accessed via a variable or expression of type `SuperC`, only methods and instance variables defined on `SuperC` are available.

- Class downcasts: If `C` is a subclass of `SuperC` and `sc` is an instance of `SuperC`, then `sc is C` will be true iff `sc` is actually an instance of `C`. (Note: Again, a downcast does not affect the in-memory representation.)

- Objective-C class casting: The rules above also apply when one of the classes in question is defined with the `@objc` attribute or inherits from an Objective-C class.

- Class casts to AnyObject: Any class reference can be cast to `AnyObject` and then cast back to the original type. See "AnyObject" below.

- If a struct or enum type conforms to `_ObjectiveCBridgeable`, then classes of the associated bridging type can be cast to the struct or enum type and vice versa. See "The `_ObjectiveCBridgeable` Protocol" below.

Invariants:

- For any two class types `C1` and `C2`: `c is C1 && c is C2` iff `((c as? C1) as? C2) != nil`
- For any class type `C`: `c is C` iff `(c as! AnyObject) is C`
- For any class type `C`: if `c is C`, then `(c as! AnyObject) as! C === c`

CoreFoundation types

- If `CF` is a CoreFoundation type, `cf` is an instance of `CF`, and `NS` is the corresponding Objective-C type, then `cf is NS == true`
- Further, since every Objective-C type inherits from `NSObject`, `cf is NSObject == true`
- In the above situation, if `T` is some other type and `cf is NS == true`, then `cf as! NS is T` iff `cf is T`.

The intention of the above is to treat instances of CoreFoundation types as being simultaneously instances of the corresponding Objective-C type, in keeping with the general dual nature of these types. In particular, if a protocol conformance is declared on the Objective-C type, then instances of the CoreFoundation type can be cast to the protocol type directly.

XXX TODO: Converse? If ObjC instance has CF equivalent and CF type is extended, ... ??

Objective-C types

The following discussion applies in three different cases:

- *Explicit* conversions from use of the `is`, `as?`, and `as!` operators.
- *Implicit* conversions from Swift to Objective-C: These conversions are generated automatically when Swift code calls an Objective-C function or method with an argument that is not already of an Objective-C type,

  or when a Swift function returns a value to an Objective-C caller.
* *Implicit* conversions from Objective-C to Swift: These are generated automatically when arguments are passed from an Objective-C caller to a Swift function, or when an Objective-C function returns a value to a Swift caller. Unless stated otherwise, all of the following cases apply equally to all three of the above cases.

Explicit casts among Swift and Objective-C class types follow the same general rules described earlier for class types in general. Likewise, explicitly casting a class instance to an Objective-C protocol type follows the general rules for casts to protocol types.

XXX TODO EXPLAIN Implicit conversions from Objective-C types to Swift types XXXX.

CoreFoundation types can be explicitly cast to and from their corresponding Objective-C types as described above.

Objective-C types and protocols

* `T` is an Objective-C class type iff `T.self is NSObject.Type`
* `P` is an Objective-C protocol iff XXX TODO XXX

The `_ObjectiveCBridgeable` Protocol

The `_ObjectiveCBridgeable` protocol allows certain types to opt into custom casting behavior. Note that although this mechanism was explicitly designed to simplify Swift interoperability with Objective-C, it is not necessarily tied to Objective-C.

The `_ObjectiveCBridgeable` protocol defines an associated reference type `_ObjectiveCType`, along with a collection of methods that support casting to and from the associated `_ObjectiveCType`. This protocol allows library code to provide tailored mechanisms for casting Swift types to reference types.

Note: The associated `_ObjectiveCType` is constrained to be a subtype of `AnyObject`; it is not limited to being an actual Objective-C type. In particular, this mechanism is equally available to the Swift implementation of Foundation on non-Apple platforms and the Objective-C Foundation on Apple platforms.

Example #1: Foundation extends the `Array` type in the standard library with an `_ObjectiveCBridgeable` conformance to `NSArray`. This allows Swift arrays to be cast to and from Foundation `NSArray` instances.

```
let a = [1, 2, 3] // Array<Int>
let b = a as? AnyObject // casts to NSArray
```

Example #2: Foundation also extends each Swift numeric type with an `_ObjectiveCBridgeable` conformance to `NSNumber`.

```
let a = 1 // Int
// After the next line, b is an Optional<AnyObject>
// holding a reference to an NSNumber
let b = a as? AnyObject
// NSNumber is bridgeable to Double
let c = b as? Double
```

Other Concrete Types

In addition to the class types described earlier, Swift has several other kinds of concrete types.

Casting between the different kinds described below (such as casting an enum to a tuple) will always fail.

Structs and Enums

You cannot cast between different concrete struct or enum types. More formally:

- If `S` and `T` are struct or enum types and `s is S == true`, then `s is T` iff `S.self == T.self`.

Struct or enum types can be cast to class types if the struct or enum implements the `_ObjectiveCBridgeable` protocol as described earlier.

Optionals

Casting to and from optional types will transparently unwrap optionals as much as necessary, including nested optional types.

- If `T` and `U` are any two types, and `t` is an instance of `T`, then `.some(t) is Optional<U> == t is U`
- Optional injection: if `T` and `U` are any two types, `t` is a non-nil instance of `T`, then `t is Optional<U> == t is U`
- Optional projection: if `T` and `U` are any two types, and `t` is an instance of `T`, then `.some(t) is U == t is U`
- Nil Casting: if `T` and `U` are any two types, then `Optional<T>.none is Optional<U> == true`

Note: For "Optional Injection" above, the requirement that `t` is a non-nil instance of `T` implies that either `T` is not an `Optional` or that `T` is `Optional` and `t` has the form `.some(u)`

Examples

```
// T and U are any two distinct types (possibly optional)
// NO is any non-optional type
let t: T
let u: U
let no: NO
// Casting freely adds/removes optional wrappers
t is Optional<T> == true
t is Optional<Optional<T>> == true
Optional<T>.some(t) is T == true
Optional<Optional<T>>.some(.some(t)) is T == true
// Non-optionals cast to optionals iff they cast to the inner type
no is Optional<U> == no is U
Optional<NO>.some(no) is Optional<U> == no is U
// Non-nil optionals cast to a different type iff the inner value does
Optional<T>.some(t) is U == t is U
Optional<Optional<T>>.some(.some(t)) is U == t is U
// Nil optionals always cast to other optional types
Optional<T>.none is Optional<U> == true
// Nil optionals never cast to non-optionals
Optional<T>.none is NO == false
```

Depth Preservation: The rules above imply that nested optionals are transparently unwrapped to arbitrary depth. For example, if an instance of `T` can be cast to type `U`, then `T????` can be cast to `U????`. This can be ambiguous if the former contains a `nil` value; casting the nil to `U????` might end up with any of the following distinct values `.none`, `.some(.none)`, `.some(.some(.none))`, or `.some(.some(.some(.none)))`.

To resolve this ambiguity, the implementation should first inspect the type of the inner `.none` value and count the optional depth. Note that "optional depth" here refers to the number of optional wrappers needed to get from the innermost non-optional type to the type of the `.none`.

1. If the target allows it, the value should inject into the target with the same optional depth as the source.
2. Otherwise, the value should inject with the greatest optional depth possible.

Examples

```
// Depth preservation
// The `.none` here has type `T?` with optional depth 1
let t1: T???? = .some(.some(.some(.none)))
// This `.none` has type `T????` with optional depth 4
let t4: T???? = .none
// Result has optional depth 1, matching source
t1 as! U???? // Produces .some(.some(.some(.none)))
t1 as! U??? // Produces .some(.some(.none))
t1 as! U?? // Produces .some(.none)
t1 as! U? // Produces .none
// Result has optional depth 2, because 4 is not possible
t4 as! U?? // Produces .none
// Remember that `as?` adds a layer of optional
// These casts succeed, hence the outer `.some`
t1 as? U???? // Produces .some(.some(.some(.some(.none))))
t1 as? U??? // Produces .some(.some(.some(.none)))
t1 as? U?? // Produces .some(.some(.none))
t1 as? U? // Produces .some(.none)
t4 as? U?? // Produces .some(.none)
```

Array/Set/Dictionary Casts

For Array, Set, or Dictionary types, you can use the casting operators to translate to another instance of the same outer container (Array, Set, or Dictionary respectively) with a different component type. Note that the following discussion applies only to these specific types. It does not apply to any other types, nor is there any mechanism to add this behavior to other types.

Example: Given an `arr` of type `Array<Int>`, you can cast `arr as? Array<Any>`. The result will be a new array where each `Int` in the original array has been individually cast to an `Any`.

However, if any component item cannot be cast, then the outer cast will also fail. For example, consider the following:

```
let a: Array<Any> = [Int(7), "string"]
a as? Array<Int> // Fails because "string" cannot be cast to `Int`
```

Specifically, the casting operator acts for `Array` as if it were implemented as follows. In particular, note that an empty array can be successfully cast to any destination array type.

```
func arrayCast<T,U>(source: Array<T>) -> Optional<Array<U>> {
  var result = Array<U>()
  for t in source {
    if let u = t as? U {
      result.append(u)
    } else {
      return nil
    }
  }
  return result
}
```

Invariants

- Arrays cast iff their contents do: If `t` is an instance of `T` and `U` is any type, then `t is U == [t] is [U]`
- Empty arrays always cast: If `T` and `U` are any types, `Array<T>() is Array<U>`

Similar logic applies to `Set` and `Dictionary` casts. Note that the resulting `Set` or `Dictionary` may have fewer items than the original if the component casting operation converts non-equal items in the source into equal items in the destination.

Specifically, the casting operator acts on `Set` and `Dictionary` as if by the following code:

```
func setCast<T,U>(source: Set<T>) -> Optional<Set<U>> {
  var result = Set<U>()
  for t in source {
    if let u = t as? U {
      result.append(u)
    } else {
      return nil
    }
  }
  return result
}

func dictionaryCast<K,V,K2,V2>(source: Dictionary<K,V>) -> Optional<Dictionary<K2,V2>>
{
  var result = Dictionary<K2,V2>()
  for (k,v) in source {
    if let k2 = k as? K2, v2 = v as? V2 {
      result[k2] = v2
    } else {
      return nil
    }
  }
  return result
}
```

Collection Casting performance and `as!`

For `as?` casts, the casting behavior above requires that every element be converted separately. This can be a particular bottleneck when trying to share large containers between Swift and Objective-C code.

However, for `as!` casts, it is the programmer's responsibility to guarantee that the operation will succeed before requesting the conversion. The implementation is allowed (but not required) to exploit this by deferring the inner component casts until the relevant item is needed. Such lazy conversion can provide a significant performance improvement in cases where the data is known (by the programmer) to be safe and where the inner component casts are non-trivial. However, if the conversion cannot be completed, it is indeterminate whether the cast request will fail immediately or whether the program will fail at some later point.

Tuples

Casting from a tuple type T1 to a tuple type T2 will succeed iff the following hold:

- T1 and T2 have the same number of elements
- If an element has a label in both T1 and T2, the labels are identical
- Each element of T1 can be individually cast to the corresponding type of T2

Functions

Casting from a function type F1 to a function type F2 will succeed iff the following hold:

- The two types have the same number of arguments
- Corresponding arguments have identical types
- The return types are identical
- If F1 is a throwing function type, then F2 must be a throwing function type. If F1 is not throwing, then F2 may be a throwing or non-throwing function type.
- F1 and F2 have the same calling convention.

Note that it is *not* sufficient for argument and return types to be castable; they must actually be identical.

Caveat: The current Swift compiler supports more general casts of function types in certain cases where the compiler has enough information to statically construct the necessary adapter function. Function types that must be cast at runtime follow the less permissive rules described above.

Existential Types

Conceptually, an "existential type" is an opaque wrapper that carries a type and an instance of that type. The various existential types differ in what kinds of types they can hold (for example, `AnyObject` can only hold reference types) and in the capabilities exposed by the container (`AnyHashable` exposes equality testing and hashing).

The key invariant for existential types `E` is the following:

- Strong existential invariant: If `t` is any instance, `U` is any type, and `t is E` then `(t as! E) as? U` produces the same result as `t as? U`

Intuitively, this simply says that if you can put an instance `t` into an existential `E`, then you can take it back out again via casting. For Equatable types, this implies that the results of the two operations here are equal to each other when they succeed. It also implies that if either of these `as?` casts fails, so will the other.

`AnyObject` and `AnyHashable` do not fully satisfy the strong invariant described above. Instead, they satisfy the following weaker version:

- Weak existential invariant: If `t` is any instance, `U` is any type, and both `t is U` and `t is E`, then `(t as! E) as? U` produces the same result as `t as? U`

Objective-C Interactions

The difference between the strong and weak existential invariants comes about because casting to `AnyObject` or `AnyHashable` can trigger Objective-C bridging conversions. The result of these conversions may support casts that the original type did not. As a result, `(t as! E)` may be castable to `U` even if `t` alone is not directly castable.

One example of this is Foundation's `NSNumber` type which conditionally bridges to several Swift numeric types. As a result, when Foundation is in scope, `Int(7) is Double == false` but `(Int(7) as! AnyObject) is Double == true`. In general, the ability to add new bridging behaviors from a single type to several distinct types implies that Swift casting cannot be transitive.

Any

Any Swift instance can be cast to the type `Any`. An instance of `Any` has no useful methods or properties; to utilize the contents, you must cast it to another type. Every type identifier is an instance of the metatype `Any.Type`.

Invariants

- If `t` is any instance, then `t is Any == true`
- If `t` is any instance, `t as! Any` always succeeds
- For every type `T` (including protocol types), `T.self is Any.Type`
- Strong existential invariant: If `t` is any instance and `U` is any type, then `(t as! Any) as? U` produces the same result as `t as? U`.

AnyObject

Any class, enum, struct, tuple, function, metatype, or existential metatype instance can be cast to `AnyObject`.

The contents of an `AnyObject` container can be accessed by casting to another type:

- Weak existential invariant: If `t` is any instance, `U` is any type, `t is AnyObject`, and `t is U`, then `(t as! AnyObject) as? U` will produce the same result as `t as? U`

Implementation Notes

`AnyObject` is represented in memory as a pointer to a reference-counted object. The dynamic type of the object can be recovered from the "isa" field of the object. The optional form `AnyObject?` is the same except that it allows null.

Reference types (class, metatype, or existential metatype instances) can be directly assigned to an `AnyObject` without any conversion.

For non-reference types -- including struct, enum, and tuple types -- the casting logic will first look for an `_ObjectiveCBridgeable` conformance that it can use to convert the source into a tailored reference type. (See "The _ObjectiveCBridgeable Protocol" below for more details.)

If bridging fails, the value will be copied into an opaque heap-allocated container that is compatible with Objective-C memory management. The sole purpose of this container is to allow arbitrary Swift types to be cast to and from `AnyObject` so they may be shared with Objective-C functions that need to call Swift code. The Objective-C code cannot use the contents of this container; it can only store the reference and pass it back into Swift functions.

For platforms that do not support an Objective-C runtime, there is an alternate implementation of the internal container type that supports the same casting operations with `AnyObject`.

XXX TODO The runtime logic has code to cast protocol types to `AnyObject` only if they are compatible with `__SwiftValue`. What is the practical effect of this logic? Does it mean that casting a protocol type to `AnyObject` will sometimes unwrap (if the protocol is incompatible) and sometimes not? What protocols are affected by this?

Error (SE-0112)

The `Error` type behaves like an ordinary existential type for casting purposes.

(See "Note: 'Self-conforming' protocols" below for additional details relevant to the `Error` protocol.)

AnyHashable (SE-0131)

For casting purposes, `AnyHashable` behaves like an existential type. It satisfies the weak existential invariant above.

However, note that `AnyHashable` does not act like an existential for other purposes. For example, its metatype is named `AnyHashable.Type` and it does not have an existential metatype.

Protocol Witness types

Any protocol definition (except those that include an `associatedtype` property or which makes use of the `Self` typealias) has an associated existential type named after the protocol.

Specifically, assume you have a protocol definition

```
protocol P {}
```

As a result of this definition, there is an existential type (also called `P`). This existential type is also known as a "protocol witness type" since it exposes exactly the capabilities that are defined by the protocol. Other capabilities of the type `T` are not accessible from a `P` instance. Any Swift instance of a concrete type `T` can be cast to the type `P` iff `T` conforms to the protocol `P`.

The contents of a protocol witness can be accessed by casting to some other appropriate type:

- Strong existential Invariant: For any protocol `P`, instance `t`, and type `U`, if `t is P`, then `t as? U` produces the same result as `(t as! P) as? U`

In addition to the protocol witness type `P`, every Swift protocol `P` implicitly defines two other types: `P.Protocol` is the "protocol metatype", the type of `P.self`. `P.Type` is the "protocol existential metatype". These are described in more detail below.

Regarding Protocol casts and Optionals

When casting an Optional to a protocol type, the optional is preserved if possible. Given an instance `o` of type `Optional<T>` and a protocol `P`, the cast request `o as? P` will produce different results depending on whether `Optional<T>` directly conforms to `P`:

- If `Optional<T>` conforms to `P`, then the result will be a protocol witness wrapping the `o` instance. In this case, a subsequent cast to `Optional<T>` will restore the original instance. In particular, this case will preserve `nil` instances.
- If `Optional<T>` does not directly conform, then `o` will be unwrapped and the cast will be attempted with the contained object. If `o == nil`, this will fail. In the case of a nested optional `T???` this will result in fully unwrapping the inner non-optional.
- If all of the above fail, then the cast will fail.

For example, `Optional` conforms to `CustomDebugStringConvertible` but not to `CustomStringConvertible`. Casting an optional instance to the first of these protocols will result in an item whose `.debugDescription` will describe the optional instance. Casting an optional instance to the second will provide an instance whose `.description` property describes the inner non-Optional instance.

Metatypes

Swift supports two kinds of metatypes: In addition to the regular metatypes supported by many other languages, it also has *existential* metatypes that can be used to access static protocol requirements.

Metatypes

For every type `T`, there is a unique instance `T.self` that represents the type at runtime. As with all instances, `T.self` has a type. We call this type the "metatype of `T`". Technically, static variables or methods of a type belong to the `T.self` instance and are defined by the metatype of `T`:

Example:

```
struct S {
  let ivar = 2
  static let svar = 1
}
S.ivar // Error: only available on an instance
S().ivar // 2
type(of: S()) == S.self
S.self.svar // 1
S.svar // Shorthand for S.self.svar
```

For most Swift types, the metatype of `T` is named `T.Type`. However, in the following cases the metatype has a different name:

- For a nominal protocol type `P`, the metatype is named `P.Protocol`
- For non-protocol existential types `E`, the metatype is also named `E.Protocol`. For example, `Any.Protocol`, `AnyObject.Protocol`, and `Error.Protocol`.
- For a type bound to a generic variable `G`, the metatype is named `G.Type` *even if* `G` *is bound to a protocol or existential type*. Specifically, if `G` is bound to the nominal protocol type `P`, then `G.Type` is another name for the metatype `P.Protocol`, and hence `G.Type.self == P.Protocol.self`.
- As explained above, although `AnyHashable` behaves like an existential type in some respects, its metatype is called `AnyHashable.Type`.

Example:

```
// Metatype of a struct type
struct S: P {}
S.self is S.Type // always true
S.Type.self is S.Type.Type // always true
let s = S()
type(of: s) == S.self // always true
type(of: S.self) == S.Type.self

// Metatype of a protocol (or other existential) type
protocol P {}
P.self is P.Protocol // always true
// P.Protocol is a metatype, not a protocol, so:
```

```
P.Protocol.self is P.Protocol.Type // always true
let p = s as! P
type(of: p) == P.self // always true

// Metatype for a type bound to a generic type variable
f(s) // Bind G to S
f(p) // Bind G to P
func f<G>(_ g: G) {
   G.self is G.Type // always true
}
```

Invariants

- For a nominal non-protocol type `T`, `T.self is T.Type`
- For a nominal protocol type `P`, `P.self is P.Protocol`
- `P.Protocol` is a singleton: `T.self is P.Protocol` iff `T` is exactly `P`
- A non-protocol type `T` conforms to a protocol `P` iff `T.self is P.Type` (If `T` is a protocol type, see "Self conforming existential types" below for details.)
- `T` is a subtype of a non-protocol type `U` iff `T.self is U.Type`
- Subtypes define metatype subtypes: if `T` and `U` are non-protocol types, `T.self is U.Type == T.Type.self is U.Type.Type`
- Subtypes define metatype subtypes: if `T` is a non-protocol type and `P` is a protocol type, `T.self is P.Protocol == T.Type.self is P.Protocol.Type`

Existential Metatypes

Protocols can specify constraints and provide default implementations for instances of types. They can also specify constraints and provide default implementations for static members of types. As described above, casting a regular instance of a type to a protocol type produces a protocol witness instance that exposes only those features required or provided by the protocol. Similarly, a type identifier instance (`T.self`) can be cast to a protocol's "existential metatype" to expose just the parts of the type corresponding to the protocol's static requirements.

The existential metatype of a protocol `P` is called `P.Type`. (Recall that for a non-protocol type `T`, the expression `T.Type` refers to the regular metatype. Non-protocol types do not have existential metatypes. For a generic variable `G`, the expression also refers to the regular metatype, even if the generic variable is bound to a protocol. There is no mechanism in Swift to refer to the existential metatype via a generic variable.)

In essence, an existential metatype simply defines an existential that can hold a metatype instance. An instance of the existential metatype `Any.Type` can hold any metatype instance. An instance of the existential metatype `P.Type` of a protocol `P` can hold a metatype instance of any type that conforms to `P`. As with other existentials, casting *from* an existential metatype is equivalent to casting the contents of the existential. Casting *to* an existential metatype succeeds whenever the source is a conforming metatype instance (or can be unwrapped to yield such a metatype instance).

Example

```
protocol P {
   var ivar: Int { get }
   static svar: Int { get }
}
struct S: P {
   let ivar = 1
```

```
    static let svar = 2
}
S().ivar // 1
S.self.svar // 2
(S() as! P).ivar // 1
(S.self as! P.Type).svar // 2
```

Invariants

- If `T` conforms to `P` and `t` is an instance of `T`, then `t is P` and `T.self is P.Type`
- If `P` is a sub-protocol of `P1` and `T` is any type, then `T.self is P.Type` implies that `T.self is P1.Type`
- Since every type `T` conforms to `Any`, `T.self is Any.Type` is always true
- Since every class type `C` conforms to `AnyObject`, `C.self is AnyObject.Type` is always true (this includes Objective-C class types)

Note: "Self conforming" existential types

As mentioned above, a protocol definition for `P` implicitly defines types `P.Type` (the existential metatype) and `P.Protocol` (the metatype). It also defines an associated type called `P` which is the type of a container that can hold any object whose concrete type conforms to the protocol `P`.

A protocol is "self conforming" if the container type `P` conforms to the protocol `P`. This is equivalent to saying that `P.self` is an instance of `P.Type`. (Remember that `P.self` is always an instance of `P.Protocol`.)

This is a concern for Swift because of the following construct, which attempts to invoke a generic `f` in a situation where the concrete instance clearly conforms to `P` but is represented as a `P` existential:

```
func f<T:P>(t: T) { .. use t .. }
let a : P = something
f(a)
```

This construct is valid only if `T` conforms to `P` when `T = P`; that is, if `P` self-conforms.

A similar situation arises with generic types:

```
struct MyGenericType<T: P> {
  init(_ value: T) { ... }
}
let a : P
let b : MyGenericType(a)
```

As above, since `a` has type `P`, this code is instantiating `MyGenericType` with `T = P`, which is only valid if `P` conforms to `P`. Note that any protocol that specifies static methods, static properties, associated types, or initializers cannot possibly be self-conforming.

Although the discussion above specifically talks about protocols, it applies equally well to other existential types. As of Swift 5.3, the only self-conforming existential types are `Any`, `Error`, and Objective-C protocols that have no static requirements.

Invariants

- `Any` self-conforms: `Any.self is Any.Type == true`

- `Error` self-conforms: `Error.self is Error.Type == true`
- If `P` self-conforms and is a sub-protocol of `P1`, then `P.self is P1.Type == true`

For example, the last invariant here implies that for any Objective-C protocol `OP` that has no static requirements, `OP.self is AnyObject.Type`. This follows from the fact that `OP` self-conforms and that every Objective-C protocol has `AnyObject` as an implicit parent protocol.

Implementation Notes

Casting operators that appear in source code are translated into SIL in a variety of ways depending on the details of the types involved. One common path renders `as!` into some variant of the `unconditional_checked_cast` instruction and renders `as?` and `is` into a conditional branch instruction such as `checked_cast_br`.

SIL optimization passes attempt to simplify these. In some cases, the result of the cast can be fully determined at compile time. In a few cases, the compiler can generate tailored code to perform the cast (for example, assigning a reference to an `AnyObject` variable). In other cases, the compiler can determine whether the cast will succeed without necessarily being able to compute the result. For example, casting a class reference to a superclass will always succeed, which may allow a `checked_cast_br` instruction to be simplified into a non-branching `unconditional_checked_cast`. Similarly, an `as?` cast that can be statically proven to always fail can be simplified into a constant `nil` value. Note that these SIL functions are also used for other purposes, including implicit bridging conversions when calling into Objective-C. This makes them common enough to be subject to some limited optimizations even in `-Onone` mode.

When SIL is translated into LLVM IR, the remaining cast operations are rendered into calls to appropriate runtime functions. The most general such function is `swift_dynamicCast` which accepts a pointer to the input value, a location in which to store the result, and metadata describing the types involved. When possible, the compiler prefers to instead emit calls to specialized variants of this function with names like `swift_dynamicCastClass` and `swift_dynamicCastMetatypeToObject` (a full list is in the Runtime Documentation). These specialized versions require fewer arguments and perform fewer internal checks, which makes them cheaper to use.

The `swift_dynamicCast` function examines the input and output types to determine appropriate conversions. This process recurses on the input type to examine the contents of an existential container or `Optional`. If the output is an existential container or `Optional` type, it will also recurse on the output type to determine a suitable base conversion and then must configure the container appropriately for the contents. It may also perform additional metadata lookups to determine whether either type conforms to `Hashable`, `Error`, or `ObjectiveCBridgeable` protocols. For collections, this process may end up recursively attempting to convert each element.

Compared to Swift 5.3

These are some of the ways in which Swift 5.3 differs from the behavior described above:

- Casts for which the target type is "more Optional" than the static source type previously produced errors. This disallowed all of the following: injecting an `Int` into an `Int?`, extracting an `Int?` from an opaque `Any` container, and casting an `Array<Int>` to an `Array<Int?>`. This document allows all of these.

```
let a = 7
// Swift 5.3: error: cannot downcast to a more optional type
// Specification: returns true
```

```
a is Int?
// Swift 5.3: error: cannot downcast to a more optional type
// Specification: returns false
a is Optional<Double>

let b: Int? = 7
let c: Any = b
// Swift 5.3: error: cannot downcast to a more optional type
// Specification: returns true
c is Int?
```

- An instance of a CoreFoundation type could sometimes be cast to a protocol defined on the companion Obj-C type and sometimes not. To make the behavior consistent, we had to choose one; having such casts always succeed seems more consistent with the general dual nature of Obj-C/CF types.

```
import Foundation
protocol P {}
extension NSString: P {}
let a = CFStringCreateWithCString(nil, "hello, world",
                CFStringBuiltInEncodings.UTF8.rawValue)
// Swift 5.3: prints "true"
print(a is P)
let b: Any = a
// Swift 5.3: prints "false"
// Specification: prints "true"
print(b is P)
```

- The Swift 5.3 compiler asserts attempting to cast a struct to AnyObject

```
struct S {}
let s = S()
// Swift 5.3: Compiler crash (in asserts build)
// Specification:  Succeeds via __SwiftValue boxing
s as? AnyObject
```

- `NSNumber()` does not cast to itself via `as?` in unoptimized builds

```
import Foundation
let a = NSNumber()
// true in 5.3 for optimized builds; false for unoptimized builds
print((a as? NSNumber) != nil)
```

- `Optional<NSNumber>` does not project

```
import Foundation
let a: Optional<NSNumber> = NSNumber()
// Swift 5.3: false
// Specification: true
print(a is NSNumber)
// Swift 5.3: nil
// Specification: .some(NSNumber())
print(a as? NSNumber)
```

- Casting `NSNumber()` to `Any` crashes at runtime

```
import Foundation
let a = NSNumber()
// Swift 5.3: Runtime crash (both optimized and unoptimized builds)
// Specification: Succeeds
print(a is Any)
```

- SR-2289: CF types cannot be cast to protocol existentials

```
import Foundation
protocol P {}
extension CFBitVector : P {
  static func makeImmutable(from values: Array<UInt8>) -> CFBitVector {
    return CFBitVectorCreate(nil, values, values.count * 8)
  }
}
// Swift 5.3: Crashes in unoptimized build, prints true in optimized build
// Specification: prints true
print(CFBitVector.makeImmutable(from: [10,20]) is P)
```

- SR-4552: Cannot cast `Optional<T> as Any` to protocol type. Note that this is a particular problem for reflection with weak fields, since `Mirror` reflects those as `Any` containing an `Optional` value.

```
protocol P {}
class C: P {}
let c: C? = C()
let a = c as? Any
// Swift 5.3: prints "false"
// Specification: prints "true"
print(a is P)
```

- SR-8964: `Any` containing `Optional<Any>` cannot cast to `Error`

```
struct MyError: Error { }
let a: Any? = MyError()
let b: Any = a
// Swift 5.3: Prints false
// Specification: prints true
print(b is Error)
```

- SR-6126: Inconsistent results for nested optionals

```
// Note: SR-6126 includes many cases similar to the following
let x: Int? = nil
print(x as Int??) // ==> "Optional(nil)"
// Swift 5.3: prints "nil"
// Specification: should print "Optional(nil)" (same as above)
print((x as? Int??)!)
```

- `Error.self` does not fully self-conform

```
// Swift 5.3: Prints "false"
// Specification: prints "true"
print(Error.self is Error.Type)
```

- Objective-C protocol metatypes do not fully self-conform

```
import Foundation
let a = NSObjectProtocol.self
print(a is NSObjectProtocol.Type)
```

- SR-1999: Cannot cast `Any` contents to a protocol type

```
protocol P {}
class Foo: P {}
let optionalFoo: Foo? = Foo()
let any: Any = optionalFoo
// Swift 5.3: Prints "false"
// Specification: prints "true"
print(any as? P)
```

- XXX TODO List others