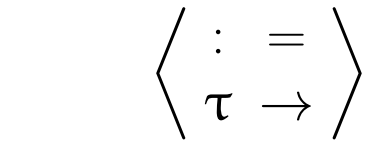


Compiling Swift Generics

Slava Pestov

October 17, 2022

Preface

This is a book about the implementation of generic programming in Swift. While it is primarily meant to be a reference for Swift compiler contributors, it should also be of interest to other language designers, type system researchers, and even just curious Swift programmers. Some familiarity with general compiler design and the Swift language is assumed. A basic understanding of abstract algebra is also helpful.

This work began as a paper about the Requirement Machine, a new implementation of the core algorithms in Swift generics which shipped with Swift 5.6. After making some progress on writing the paper, I realized that a reference guide for the entire generics implementation would be more broadly useful to the community. I worked backwards, adding more preliminary material and revising subsequent sections until reaching a fixed point, hopefully converging on something approximating a coherent and self-contained treatment of this cross-section of the compiler.

Part I of this book outlines the basic building blocks. Each chapter in the first part depends on the previous chapters; a determined (or stubborn) reader should be able to work through them sequentially, however you might find it easier to skim some sections and refer back later.

Part II details how various language features are built up from the core concepts of generics. In the second part, each chapter is mostly independent of the others.

Part III dives into the Requirement Machine, which implements generic signature queries and requirement minimization. This is the most technical part of the book.

Occasional historical asides explain when major features were introduced, citing the relevant Swift evolution proposals. The bibliography lists all cited proposals. There is also an automatically-generated index at the end; you might find it useful for looking up unfamiliar terminology.

The Swift compiler is implemented in C++. To help separate essential from incidental complexity, concepts are described without immediately referencing the source code. Every chapter ends with a "Source Code Reference" section, structured somewhat like an API reference, which translates what was previously explained into code. You can skip this material if you're not interested in the practicalities of the compiler implementation itself. No knowledge of C++ is required outside of these sections.

This book was typeset with TEX. You can find the latest version in our git repository:

https://github.com/apple/swift/tree/main/docs/Generics

Contents

Part I.

Nuts and Bolts

1. Introduction

Swift generics were designed with four primary goals in mind:

1. Generic definitions should be independently type checked, without knowledge of all possible concrete type substitutions that they are invoked with.

2. Shared libraries that export generic definitions should be able to evolve resiliently without requiring recompilation of clients.

3. Layouts of generic types should be determined by their concrete substitutions, with fields of generic parameter type stored inline.

4. Abstraction over concrete types with generic parameters should only impose a cost across module boundaries, or in other situations where type information is not available at compile time.

The Swift compiler achieves these goals as follows:

1. The interface between a generic definition and its uses is mediated by **generic requirements**. The generic requirements describe the behavior of the generic parameter types inside the function body, and the generic arguments at the call site are checked against the declaration's generic requirements at compile time.

2. Generic functions receive **runtime type metadata** for each generic argument from the caller. Type metadata defines operations to abstractly manipulate values of their type without knowledge of their concrete layout.

3. Runtime type metadata is constructed for each type in the language. The **runtime type layout** of a generic type is computed recursively from the type metadata of the generic arguments. Generic types always store their contents without boxing or indirection.

4. The optimizer can generate a **specialization** of a generic function in the case where the definition is visible at the call site. This eliminates the overhead of runtime type metadata and abstract value manipulation.

An important part of compiler implementation is the design of domain objects to model concepts in the language being compiled. One way to think of a compiler is that

it is *a library for implementing the target language*. A well-designed set of domain objects facilitates the introduction of new language features that compose existing functionality in new ways.

The generics implementation deals with four fundamental domain objects: *generic signatures*, *substitution maps*, *requirement signatures*, and *conformances*. As you will see, they are defined as much by their inherent structure, as their relationship with each other. Subsequent chapters will dive into all the details, but first, we're going to look at a series of worked examples to help you understand the big picture.

1.1. Generic Functions

Consider these two rather contrived function declarations:

```
1 func identity(_ x: Int) -> Int { return x }
2 func identity(_ x: String) -> String { return x }
```

Apart from the parameter and return type, both have the same exact definition, and indeed you can write the same function for any concrete type. Your aesthetic sense might lead you to replace both with a single generic function:

```
1 func identity<T>(_ x: T) -> T { return x }
```

While this function declaration is trivial, it illustrates some important concepts and allows us to introduce terminology. You'll see a full description of the compilation pipeline in the next chapter, but for now, let's consider a simplified view where we begin with parsing, then type checking, and finally code generation.

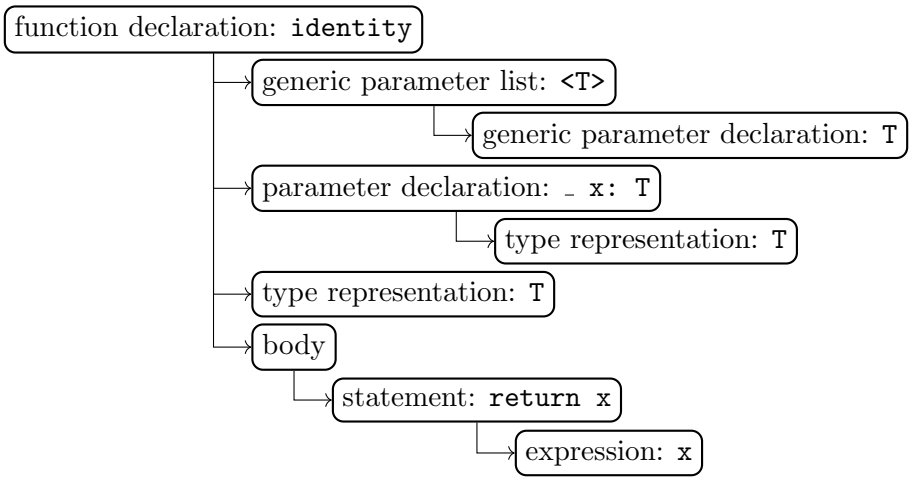
Parsing Figure 1.1 shows the abstract syntax tree produced by the parser before type checking. The key elements:

1. The *generic parameter list* `<T>` introduces a single *generic parameter declaration* named `T`. As its name suggests, this declares the generic parameter type `T`, scoped to the entire source range of this function.

2. The *type representation* `T` appears twice, first in the the parameter declaration `_ x: T` and then as return type of `identity(_:)`. A type representation is the purely syntactic form of a type. The parser does not perform name lookup, so the type representation stores the identifier `T` and does not refer to the generic parameter declaration of `T` in any way.

3. The function body contains an expression referencing `x`. Again, the parser does not perform name lookup, so this is just the identifier `x` and is not associated with the parameter declaration `_ x: T`.

Figure 1.1.: The abstract syntax tree for `identity(_:)`



Type checking Some additional structure is formed during type checking:

1. The generic parameter declaration `T` declares the generic parameter type `T`. Types are distinct from type declarations in Swift; some types denote a *reference* a type declaration, and some are *structural* (such as function types or tuple types).

2. The type checker constructs a *generic signature* for our function declaration. The generic signature has the printed representation `<T>` and contains the single generic parameter type `T`. This is the simplest possible generic signature, apart from the empty generic signature of a non-generic declaration.

3. The type checker performs *type resolution* to transform the type representation `T` appearing in our parameter declaration and return type into a semantic *type*. Type resolution queries name lookup for the identifier `T` at the source location of each type representation, which finds the generic parameter declaration `T` in both cases. This type declaration declares the generic parameter type `T`, which becomes the resolved type.

4. There is now enough information to form the function's *interface type*, which is the type of a reference to this function from expression context. The interface type of a generic function declaration is a *generic function type*, composed from the function's generic signature, parameter types, and return type:

   ```
   <T> (T) -> T
   ```

The final step is the type checking of the function's body. The expression type checker queries name lookup for the identifier `x`, which finds the parameter declaration `_ x: T`.

While the type of our function parameter is the generic parameter type `T`, inside the body of a generic function it becomes a different kind of type, called a *primary archetype*. The distinction isn't terribly important right now, and it will be covered in Chapter 9. It suffices to say that we'll use the notation ⟦`T`⟧ for the primary archetype corresponding to the generic parameter type `T`.

With that out of the way, the expression type checker assigns the type ⟦`T`⟧ to the expression `x` appearing in the return statement. As expected, this matches the declared return type of the function.

Code generation We've now successfully type checked our function declaration. How might we generate code for it? Recall the two concrete implementations that we folded into our single generic function:

```
1 func identity(_ x: Int) -> Int { return x }
2 func identity(_ x: String) -> String { return x }
```

The calling conventions of these functions differ significantly:

1. The first function receives and returns the `Int` value in a machine register. The `Int` type is *trivial*,[1] meaning it can be copied and moved at will.

2. The second function is trickier. A `String` is stored as a 16-byte value in memory, and contains a pointer to a reference-counted buffer. When manipulating values of a non-trivial type like `String`, memory ownership comes into play.

   The standard ownership semantics for a Swift function call are defined such that the caller retains ownership over the parameter values passed into the callee, while the callee transfers ownership of the return value to the caller. This means that the `identity(_:)` function cannot just return the value `x`; instead, it must first create a logical copy of `x` that it owns, and then return this owned copy. This is achieved by incrementing the string value's buffer reference count via a call to a runtime function.

More generally, every Swift type has a size and alignment, and defines three fundamental operations that can be performed on all values of that type: moving the value, copying the value, and destroying the value. A move is semantically equivalent to, but more efficient than, copying a value followed by destroying the old copy.[2]

With a trivial type, moving or copying a value simply copies the value's bytes from one memory location to another, and destroying a value does nothing. With a reference

[1] Or POD, for you C++ folks.

[2] Of course if move-only types are ever introduced into the language, this will no longer be so; a new kind of value will exist which cannot be copied.

type, these operations update the reference count. Copying a reference increments the reference count on its heap-allocated backing storage, and destroying a reference decrements the reference count, deallocating the backing storage when the reference count reaches zero. Even more complex behaviors are also possible; a struct might contain a mix of trivial types and references, for example. Weak references and existential types also have non-trivial value operations.

As the joke goes, every problem in computer science can be solved with an extra level of indirection. The calling convention for a generic function takes *runtime type metadata* for every generic parameter in the function's generic signature. Every type in the language has a reified representation as runtime type metadata, storing the type's size and alignment together with function pointers implementing the move, copy and destroy operations. The generated code for a generic function abstractly manipulates values of generic parameter type using the runtime type metadata provided by the caller. An important property of runtime type metadata is *identity*; two pointers to runtime type metadata are equal if and only if they represent the same type in the language.

More details

- Types: Chapter 3
- Function declarations: Section 4.2
- Generic parameter lists: Chapter 5
- Type resolution: Chapter 10

Substitution maps Let us now turn our attention to the callers of generic functions. A *call expression* references a *callee* together with a list of arguments. The callee is some other expression with a function type. Some possible callees include references to named function declarations, type expressions (which invokes a constructor), function parameters and local variables of function type, and results of other calls which return functions. In our example, we might call the `identity(_:)` function as follows:

```
1 identity(3)
2 identity("Hello, Swift")
```

The callee here is a direct reference to the declaration of `identity(_:)`. In Swift, calls to generic functions never specify their generic arguments explicitly; instead, the type checker infers them from the types of call argument expressions. A reference to a named generic function stores a *substitution map* mapping each generic parameter type of the callee's generic signature to the inferred generic argument, also called the *replacement type*.

The generic signature of `identity(_:)` has a single generic parameter type. The two references to `identity(_:)` have different substitution maps; the first substitution map has the replacement type `Int`, and the second `String`. We will use the following notation for these substitution maps:

Types
`T := Int`

Types
`T := String`

We can apply a substitution map to the interface type of our function declaration to get the *substituted type* of the callee:

`<T> (T) -> T` × | **Types** | = `(Int) -> Int`
| --- |
| `T := Int` |

Substitution maps also play a role in code generation. When generating a call to a generic function, the compiler emits code to realize the runtime type metadata for each replacement type in the substitution map. The types `Int` and `String` are *nominal types* defined in the standard library. These types are non-generic and have a fixed layout, so their runtime type metadata can be recovered by taking the address of a constant symbol exported by the standard library.

Structural types are slightly more complicated. Suppose we were instead compiling a call to `identity(_:)` where the replacement type for `T` was some function type, say `(Int, String) -> Float`. Function types can have arbitrary parameter and return types. Therefore, structural type metadata is *instantiated* by calling one of several *metadata access functions*, declared in the runtime. These runtime entry points take metadata for the parameter types and return type, construct metadata representing the function type, and cache the result for future accesses.

More details

- Substitution maps: Chapter 7

Specialization The passing of runtime type metadata and the resulting indirect manipulation of values incurs a performance penalty. As an alternative, if the definition of a generic function is visible at the call site, the optimizer can generate a *specialization* of the generic function by cloning the definition and applying the substitution map to all types appearing in the function's body. Definitions of generic functions are always visible to the specializer within their defining module. Shared library developers can also opt-in to exporting the body of a function across module boundaries with the `@inlinable` attribute.

More details

- `@inlinable` attribute: Section 2.5

1.2. Generic Types

For our next example, consider this simple generic struct storing two values of the same type:

```
struct Pair<T> {
  let first: T
  let second: T

  init(first: T, second: T) {
    self.first = first
    self.second = second
  }
}
```

This struct declaration contains three members: two stored property declarations, and a constructor declaration. Recall that declarations have an *interface type*, which is the type of a reference to the declaration from expression context. The interface type of `first` and `second` is the generic parameter type `T`.

When a type declaration is referenced from expression context the result is a value representing the type, and the type of this value is a metatype type, so the interface type of `Pair` is the metatype type `Pair<T>.Type`.

Type declarations also have a more primitive notion of a *declared interface type*, which is the type assigned to a reference to the declaration from type context. The declared interface type of `Pair` is the *generic nominal type* `Pair<T>`. The interface type of a type declaration is the metatype of its declared interface type.

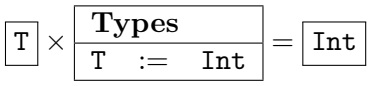
Instances of `Pair` store their fields inline without boxing, and the layout of `Pair` depends on the generic parameter `T`. If you declare a local variable whose type is the generic nominal type `Pair<Int>`, the compiler can directly compute the type's layout to determine the size of the stack allocation:

```
let twoIntegers: Pair<Int> = ...
```

To compute the layout, the compiler first factors the type `Pair<Int>` into the application of a substitution map to the declared interface type:

`Pair<Int>`	=	`Pair<T>`	×	**Types**
				`T` := `Int`

The compiler then computes the substituted type of each stored property by applying this substitution map to each stored property's interface type:



Therefore both fields of `Pair<Int>` have a substituted type of `Int`. The `Int` type has a size of 8 bytes and an alignment of 8 bytes, from which we derive that `Pair<Int>` has a size of 16 bytes and alignment of 8 bytes.

However, the layout is not always known at compile time, in which case we need the runtime type metadata for `Pair<T>`. When compiling the declaration of `Pair`, the compiler emits a *metadata access function* which takes the type metadata for `T` as an argument. The metadata access function calculates the layout of `Pair<T>` for this `T` with the same algorithm as the compiler, but at runtime, and caches the result.

Note that the runtime type metadata for `Pair<Int>` has two parts:

1. A common prefix present in all runtime type metadata, which includes the total size and alignment of a value.
2. A private area specific to the declaration of `Pair<T>`, such as the *field offset vector* storing the starting offset of each field within a value.

The first part comes into play if we call our `identity(_:)` function with a value of type `Pair<Int>`. The generated code for the call invokes a metadata access function for `Pair<T>` with the metadata for `Int` as an argument, and passes the resulting metadata for `Pair<Int>` to `identity(_:)`. The implementation of `identity(_:)` doesn't know that it is dealing with a `Pair<Int>`, but it uses the provided metadata to abstractly manipulate the value.

The second part is used by the constructor implementation. The constructor does not have a generic parameter list of its own, but it is nested inside of a generic type, so it inherits the generic signature of the type, which is `<T>`. The interface type of this constructor is the generic function type:

```
<T> (T, T) -> Pair<T>
```

Recall our declaration of the `twoIntegers` variable. Let's complete the declaration by writing down an initial value expression which calls the constructor:

```
1 let twoIntegers: Pair<Int> = Pair(first: 1, second: 2)
```

At the call site, we have full knowledge of the layout of `twoIntegers`. However, the implementation of `Pair.init` only knows that it is working with a `Pair<T>`, and not a `Pair<Int>`. The generated code for the constructor calls the metadata access function

for `Pair<T>` with the provided metadata for `T`. Since it knows it is working with a `Pair<T>`, it can look inside the private area to get the field offset of `first` and `second`, and assign the two parameters into the `first` and `second` stored properties of `self`.

More details

- Type declarations: Section 4.1
- Context substitution map: Section 7.1

1.3. Protocols

Our `identity(_:)` function and the `Pair` type did not state any generic requirements, so they couldn't do much with their generic values except pass them around, which the compiler expresses in terms of the fundamental value operations—move, copy and destroy.

We can do more interesting things with our generic parameter types by writing down generic requirements. The most important kind is the *protocol conformance requirement*, which states that the replacement type for a generic requirement must conform to the given protocol.

```
protocol Shape {
  func draw()
}

func drawShapes<S: Shape>(_ shapes: [S]) {
  for shape in shapes {
    shape.draw()
  }
}
```

The `drawShapes(_:)` function takes an array of values whose type conforms to `Shape`. You can also write the declaration of `drawShapes(_:)` using a trailing `where` clause, or avoid the explicit generic parameter list altogether and declare an *opaque parameter type* instead:

```
func drawShapes<S>(_ shapes: [S]) where S: Shape
func drawShapes(_ shapes: [some Shape])
```

The generic signatures we've seen previously were rather trivial, only storing a single generic parameter type. More generally, a generic signature actually consists of a list of

generic parameter types together with a list of requirements. Irrespective of the surface syntax, the generic signature of `drawShape(_:)` will have a single requirement. We will use the following notation for generic signatures with requirements:

```
<S where S: Shape>
```

The interface type of `drawShapes(_:)` is a generic function type incorporating this generic signature:

```
<S where S: Shape> (S) -> ()
```

Inside the body of `drawShapes(_:)`, the `shape` local variable bound by the `for` loop is a value of type ⟦`S`⟧ (remember, generic parameter types become archetype types inside the function body; but as before, the distinction doesn't matter right now). Since `S` is subject to the conformance requirement `S: Shape`, we can call the `draw()` method of the `Shape` protocol on `shape`. More precisely, a *qualified lookup* of the identifier `draw` with a base type of ⟦`S`⟧ will find the `draw()` method of `Shape` as a consequence of the conformance requirement.

How does the compiler generate code for the call `shape.draw()`? Once again, we need to introduce some indirection. For each conformance requirement in the generic signature of a generic function, the generic function receives a *witness table* from the caller. The layout of a witness table is determined by the protocol's requirements; a method becomes an entry storing a function pointer. To call our protocol method, the compiler loads the function pointer from the witness table, and invokes it with the argument value of `shape`.

Note that `drawShapes(_:)` operates on a homogeneous array of shapes. While the array contains an arbitrary number of elements, `drawShapes(_:)` only receives a single runtime type metadata for `S`, and one witness table for the conformance requirement `S: Shape`, which together describe all elements of the array.

More details

- Protocols: Section 5.4
- Constraint types: Section 5.1
- Trailing `where` clauses: Section 5.2
- Opaque parameters: Section 5.3
- Name lookup: Section 2.1

Conformances We can write a struct declaration conforming to `Shape`:

```
struct Circle: Shape {
  let radius: Double
  func draw() {...}
}
```

The declaration of `Circle` states a *conformance* to the `Shape` protocol in its inheritance clause. The type checker constructs an object called a *normal conformance*, which records the mapping from the protocol's requirements to the members of the conforming type which *witness* those requirements.

When the compiler generates the code for the declaration of `Circle`, it emits a witness table for each normal conformance defined on the type declaration. In our case, there is just a single requirement `Shape.draw()`, witnessed by the method `Circle.draw()`. The witness table for this conformance references the witness (indirectly, because the witness is always wrapped in a *thunk*, which is a small function which shuffles some registers around and then calls the actual witness. This must be the case because protocol requirements use a slightly different calling convention than ordinary generic functions).

Now, let's look at a call to `drawShape(_:)` with an array of circles:

```
drawShapes([Circle(radius: 1), Circle(radius: 2)])
```

Recall that a reference to a generic function declaration comes with a substitution map. Substitution maps store a replacement type for each generic parameter of a generic signature, so our substitution map maps `S` to the replacement type `Circle`. When the generic signature has conformance requirements, the substitution map also stores a conformance for each conformance requirement. This is the "proof" that the concrete replacement type actually conforms to the protocol.

The type checker finds conformances by *global conformance lookup*. The call to `drawShape(_:)` will only type check if the replacement type conforms to `Shape`; the type checker rejects a call that provides an array of integers for example, because there is no conformance of `Int` to `Shape`.[3]

We will use the following notation for substitution maps storing a conformance:

Types
`S := Circle`

Conformances
`Circle: Shape`

When emitting code to call to a generic function, the compiler looks at the substitution map and emits a reference to runtime type metadata for each replacement type, and a

[3] Of course, you could define this conformance with an extension.

reference to the witness table for each conformance. In our case, `drawShapes(_:)` takes a single runtime type metadata and a single witness table for the conformance. (The contents of the witness table were emitted when compiling the declaration of `Circle`; compiling the substitution map references this existing witness table.)

More details

- Conformances: Chapter 8
- Conformance lookup: Section 8.1

Associated types Perhaps the simplest example of a protocol with an associated type is the `Iterator` protocol in the standard library. This protocol abstracts over an iterator which produces elements of a type that depends on the conformance:

```
protocol IteratorProtocol {
  associatedtype Element
  mutating func next() -> Element?
}
```

Consider a generic function which returns the first element produced by an iterator:

```
func firstElement<I: IteratorProtocol>(_ iter: inout I) -> I.Element {
  return iter.next()!
}
```

The return type of our function is the *identifier type representation* `I.Element` with two components, "`I`" and "`Element`". Type resolution resolves this type representation to a type by performing a qualified lookup of `Element` on the base type `I`. The generic parameter type `I` is subject to a conformance requirement, and qualified lookup finds the associated type declaration `Element`.

The resolved type is a *dependent member type* composed from the generic parameter type `I` and associated type declaration `Element`. We will denote this dependent member type as `I.[IteratorProtocol]Element` to make explicit the fact that a name lookup has resolved the identifier `Element` to an associated type.

The interface type of `firstElement(_:)` is therefore this generic function type:

```
<I where I: IteratorProtocol>
(inout I) -> I.[IteratorProtocol]Element
```

More details

- Identifier type representations: Section 10.1

Type parameters A *type parameter* in some fixed generic signature is either a generic parameter type, or a dependent member type whose base type conforms to the protocol of this associated type. The generic signature of `firstElement(_:)` has two valid type parameters:

```
I
I.[IteratorProtocol]Element
```

As with generic parameter types, dependent member types become primary archetypes in the body of a generic function; we can reveal a little more about the structure of primary archetypes now, and say that a primary archetype packages a type parameter together with a generic signature.

Inside the body of `firstElement(_:)`, the result of the call expression `iter.next()!` is the optional type ⟦`I.Element`⟧`?`, which is force-unwrapped to yield the archetype type ⟦`I.Element`⟧. To manipulate a value of the element type abstractly, the compiler must be able to recover its runtime type metadata.

While metadata for generic parameters is passed in directly, for dependent member types the metadata is recovered from one or more witness tables provided by the caller. A witness table for a conformance to `IteratorProtocol` stores two entries, one for each of the protocol's requirements:

- A metadata access function to witness the `Element` associated type.
- A function pointer to witness the `next()` protocol requirement.

Type witnesses When a concrete type conforms to a protocol, the normal conformance stores a *type witness* for each of the protocol's associated types; this information is populated by the type checker during conformance checking.

Listing 1.1 shows a type that conforms to `IteratorProtocol` by producing an infinite stream of incrementing integers. Here, the associated type `Element` is witnessed by a type alias declaration with an underlying type of `Int`. This matches the return type of `NaturalNumbers.next()`. Indeed, we can omit the type alias entirely in this case, and instead rely on *associated type inference* to derive it from the interface type of the witness.

Suppose we call `firstElement(_:)` with a value of type `NaturalNumbers`:

```
1 var iter = NaturalNumbers()
2 print(firstElement(&iter))
```

Listing 1.1.: Iterator producing the natural numbers

```
struct NaturalNumbers: IteratorProtocol {
  typealias Element = Int
  var x = 0

  mutating func next() -> Int? {
    defer { x += 1 }
    return x
  }
}
```

The substitution map for the call stores the replacement type `NaturalNumbers` and the conformance of `NaturalNumbers` to `IteratorProtocol`:

Types
`I` := `NaturalNumbers`

Conformances
`NaturalNumbers: IteratorProtocol`

To compute the substituted type of the call, we apply our substitution map to the interface type of `firstElement(_:)`. Substitution transforms the parameter type `I` to the replacement type `NaturalNumbers`.

To compute the substituted return type for `I.[IteratorProtocol]Element`, we can look up the type witness in the conformance stored in the substitution map. This is entirely analogous to how the generated code for our function is able to recover the runtime type metadata for this dependent member type from a witness table at run time.

The normal conformance of `NaturalNumbers: IteratorProtocol` can be found in the substitution map, and it stores the type witness for `Element`, which is `Int`. The substituted return type is `Int`, and the substituted function type for the call is therefore:

```
(inout NaturalNumbers) -> Int
```

More details

- Type witnesses: Section 8.3
- Dependent member type substitution: Section 8.4

Associated conformances Protocols can also impose requirements on their associated types. The `Sequence` protocol in the standard library is one such example:

```
protocol Sequence {
  associatedtype Element
  associatedtype Iterator: IteratorProtocol
    where Element == Iterator.Element

  func makeIterator() -> Iterator
}
```

There are two requirements here:

1. The conformance requirement `Iterator: IteratorProtocol`, which is written as a constraint type in the inheritance clause of the `Iterator` associated type.

2. The same-type requirement `Element == Iterator.Element`, written in a trailing `where` clause.

Requirements on the generic parameters of a generic function or generic type are collected in the declaration's generic signature. A protocol analogously has a *requirement signature* which collects the requirements imposed on its associated types. A protocol always declares a single generic parameter named `Self`, and our notation for a requirement signature looks like a generic signature over the protocol `Self` type:

```
<Self where Self.[Sequence]Element ==
                Self.[Sequence]Iterator.[IteratorProtocol]Element,
            Self.[Sequence]Iterator: IteratorProtocol>
```

The conformance requirement `Self.[Sequence]Iterator: IteratorProtocol` is an *associated conformance requirement*, and associated conformance requirements appear in protocol witness tables. Therefore a witness table for a conformance to `Sequence` has *four* entries:

1. A metadata access function to witness the `Element` associated type.

2. A metadata access function to witness the `Iterator` associated type.

3. A witness table access function to witness the associated conformance requirement `Iterator: IteratorProtocol`.

4. A function pointer to the witness the `makeIterator()` protocol requirement.

Abstract conformances Let's define a `firstElementSeq(_:)` function which operates on a sequence.[4] We can call the `makeIterator()` protocol requirement to create an iterator for our sequence, and then hand off this iterator to the `firstElement(_:)` function we defined previously:

```
1 func firstElementSeq<S: Sequence>(_ sequence: S) -> S.Element {
2   var iter = sequence.makeIterator()
3   return firstElement(&iter)
4 }
```

The substitution map for the call to `firstElement(_:)` is interesting. The argument `iter` has the type ⟦`S.Element`⟧, which becomes the replacement type for the generic parameter `I` of `firstElement(_:)`. Recall that this substitution map also needs to store a conformance. Since the conforming type is an archetype and not a concrete type, global conformance lookup returns an *abstract conformance*. So our substitution map looks like this:

Types
`I` := ⟦`S.Iterator`⟧

Conformances
⟦`S.Iterator`⟧`: IteratorProtocol`

When generating code for the call, we need to emit runtime type metadata for `I` as well as a witness table for `I: IteratorProtocol`. Both of these are recovered from the witness table for the conformance `S: Sequence` that was passed by the caller of `firstElementSeq(_:)`:

1. The replacement type for `I` is ⟦`S.Iterator`⟧. Runtime type metadata for this type is recovered by calling the metadata access function for the `Iterator` associated type stored in the `S: Sequence` witness table.

2. The conformance for `I: Iterator` is an abstract conformance. We know the type ⟦`S.Iterator`⟧ conforms to `IteratorProtocol` because the `Sequence` protocol says that it does. Therefore, the witness table for this conformance is recovered by calling the witness table access function for the `Iterator: IteratorProtocol` associated conformance in our `S: Sequence` witness table.

Recall that the shape of the substitution map is determined by the generic signature of the callee. In our earlier examples, the replacement types and conformances were fully concrete, which allowed us to emit runtime type metadata and witness tables for a call by referencing global symbols.

[4] We could give both functions the same name and take advantage of function overloading, but for clarity we're not going to do that.

More generally, the replacement types and conformances are defined in terms of the type parameters of the caller's generic signature. This makes sense, because we start with the runtime type metadata and witness tables received by the caller, from which we recover the runtime metadata and witness tables required by the callee. Here, the caller is `firstElementSeq(_:)` and the callee is `firstElement(_:)`.

1.4. Language Comparison

Swift generics occupy a unique point in the design space, which avoids some of the tradeoffs inherent in the design of other popular languages:

- C++ templates do not allow for separate compilation and type checking. When a template declaration is compiled, only minimal semantic checks are performed and no code is actually generated. The body of a template declaration must be visible at each expansion point, and full semantic checks are performed after template expansion. There is no formal notion of requirements on template parameters; at a given expansion point, template expansion either succeeds or fails depending on how the substituted template parameters are used in the body of the template.

- Rust generics are separately type checked with the use of generic requirements. Unlike C++, specialization is not part of the semantic model of the language, but it is mandated by the implementation because Rust does not define a calling convention for unspecialized generic code. After type checking, the compiler completely specializes all usages of generic definitions for every set of provided generic arguments.

- Java generics are separately type checked and compiled. Only reference types can be used as generic arguments; primitive value types must be boxed on the heap. The implementation strategy uses a uniform runtime layout for all generic types, and generic argument types are not reified at runtime. This avoids the complexity of generic type layout at the virtual machine level, but it comes at the cost of runtime type checks and heap allocation.

We can summarize this with a table.

	C++	Rust	Java	Swift
Separate compilation	×	×	✓	✓
Specialization	✓	✓	×	✓
Generic requirements	×	✓	✓	✓
Unboxed values	✓	✓	×	✓

2. Compilation Model

Most developers interact with the Swift compiler through Xcode and the Swift package manager, but for simplicity let's just consider direct invocation of `swiftc` from the command line. You can invoke `swiftc`, passing a list of all source files in your module as command line arguments:

```
1 $ swiftc m.swift v.swift c.swift
```

The `swiftc` command runs the *Swift driver*. By default, the driver emits an executable. When building frameworks (or libraries, if you're not versed in Apple jargon), the driver is invoked with the `-emit-library` and `-emit-module` flags, which generate a shared library and binary module file instead. Binary modules are consumed by the compiler when importing the framework, and are discussed in Section 2.5.

Executables must define a *main function*, which is the entry point invoked when the executable is run. There are three mechanisms for doing so:

1. If the module consists of a single source file, or if there are multiple source files and one of them is named `main.swift`, then this file becomes the *main source file* of the module. The main source file can contain statements at the top level, outside of a function body; consecutive top-level statements are collected into *top-level code declarations*. The main function executes the statements of each top-level code declaration in order. Source files other than the main source file cannot contain top-level code declarations.

2. If a struct, enum or class declaration is annotated with the `@main` attribute, the declaration must contain a static method named `main()`; this method becomes the main entry point. This attribute was introduced in Swift 5.3 [1].

3. The `@NSApplicationMain` and `@UIApplicationMain` attributes are an older way to specify the main entry point on Apple platforms. When applied to a class adopting the `NSApplicationMain` or `UIApplicationMain` protocol, a main entry point is generated which calls the `NSApplicationMain()` or `UIApplicationMain()` system framework function.

The Swift driver schedules *frontend jobs* to perform the actual compilation work. Each frontend job runs the *Swift frontend* process, which is what compiler developers think

of as "the compiler." Multiple frontend jobs can run in parallel, leveraging multi-core concurrency. By default, the number of concurrent frontend jobs is determined by the number of CPU cores; this can be overridden with the `-j` driver flag. If there are more frontend jobs than can be run simultaneously, the driver queues them and kicks them off as other frontend jobs complete.

Source files are divided among frontend jobs according to the *compilation mode*:

1. In *batch mode*, the list of source files is partitioned into fixed-size batches, up to the maximum batch size. Each frontend job compiles the source files of a single batch. This is the default.

2. In *single file mode*, there is frontend job per source file, which is effectively the same as batch mode with a maximum batch size of one. Single file mode is only used for debugging and performance testing the compiler itself. The `-disable-batch-mode` command line flag instructs the driver to run in single file mode.

3. In *whole module optimization mode*, there is no parallelism; a single frontend job is scheduled to build all source files. This trades build time for quality of generated code, because the compiler is able to perform more aggressive optimization across source file boundaries. The `-wmo` driver flag enables whole module optimization.

The Swift frontend itself is single-threaded, therefore a source file is the minimum unit of parallelism.

In batch mode and single file mode, the driver can also perform an *incremental build* by re-using the result of previous compilations, providing an additional compile-time speedup. Incremental builds are described in Section 2.3.

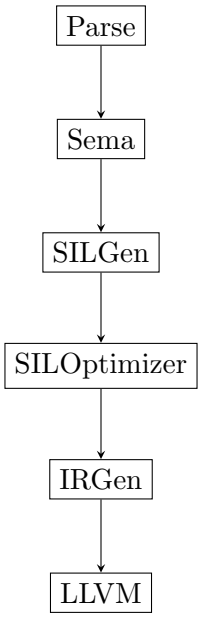
The driver invokes the frontend with a list of *primary files* and *secondary files*. The primary files are those that this specific frontend job is tasked with building, and the secondary files are the remaining source files in the module. Each source file is a primary file of exactly one frontend job, and each frontend job's primary files and secondary files together form the full list of source files in the module.

The `-###` flag driver flags performs a "dry run" which prints all commands to run without actually doing anything.

```
$ swiftc m.swift v.swift c.swift -###
swift-frontend -frontend -c -primary-file m.swift v.swift c.swift ...
swift-frontend -frontend -c m.swift -primary-file v.swift c.swift ...
swift-frontend -frontend -c m.swift v.swift -primary-file c.swift ...
ld m.o v.o c.o -o main
```

In the above, we're performing a batch mode build, but the module only has three source files, so for maximum parallelism each batch consists of a single source file. Therefore,

Figure 2.1.: The compilation pipeline



each frontend job has a single primary file, with the other two source files becoming the secondary files for the job. The final command is the linker invocation, which combines the output of each frontend job into our binary executable.

The frontend implements a classic multi-stage compiler pipeline, shown in Figure 2.1:

- **Parse:** First, all source files are parsed into an abstract syntax tree.
- **Sema:** Semantic analysis type-checks and validates the abstract syntax tree.
- **SILGen:** The type-checked syntax tree is lowered to *raw* SIL.
- **SILOptimizer:** The raw SIL is transformed into *canonical* SIL by a series of *mandatory passes*, which analyze the control flow graph and emit diagnostics; for example, *definite initialization* ensures that all storage locations are initialized.

  When the `-O` command line flag is specified, the canonical SIL is further optimized by a series of *performance passes* with the goal of improving run-time performance and reducing code size.
- **IRGen:** The optimized SIL is then transformed into LLVM IR.
- **LLVM:** Finally, the LLVM IR is handed off to LLVM, which performs various lower level optimizations before generating machine code.

Each pipeline phase can emit warnings and errors. The parser attempts to recover from errors; the presence of parse errors does not prevent Sema from running. On the other hand, if Sema emits errors, compilation stops; SILGen does not attempt to lower an invalid abstract syntax tree to SIL.

The pipeline will be slightly different depending on what the driver and frontend were asked to produce. When the frontend is instructed to emit a binary module file only, and not an object file, compilation stops after the SIL optimizer. When generating a textual interface file or TBD file, compilation stops after Sema. (Textual interfaces are discussed in Section 2.5. A TBD file is a list of symbols in a shared library, which can be consumed by the linker and is faster to generate than the shared library itself; we're not going to talk about them here.)

Various command-line flags print the output of each phase to the terminal (or some other file in conjunction with the `-o` flag), useful for debugging the compiler:

- `-dump-parse` prints the parsed syntax tree as an s-expression.[1]
- `-dump-ast` prints the type-checked syntax tree as an s-expression.
- `-print-ast` prints the type-checked syntax tree in a form that approximates what was written in source code. This is useful for getting a sense of what declarations the compiler synthesized, for example for derived conformances to protocols like `Equatable`.
- `-emit-silgen` prints the raw SIL output by SILGen.
- `-emit-sil` prints the canonical SIL output by the SIL optimizer. To see the output of the performance pipeline, also pass `-O`.
- `-emit-ir` prints the LLVM IR output by IRGen.
- `-S` prints the assembly output by LLVM.

Some command-line flags, such as those listed above, are understood by both the driver and the frontend. Certain other flags used for compiler development and debugging and only known to the frontend.

If the driver is invoked with the `-frontend` flag as the first command line flag, then instead of scheduling frontend jobs, the driver spawns a single frontend job, passing it the rest of the command line without further processing:

```
1 $ swiftc -frontend -typecheck -primary-file a.swift b.swift
```

[1] The term comes from Lisp. An s-expression represents a tree structure as nested parenthesized lists; e.g. `(a (b c) d)` is a node with three children `a`, `(b c)` and `d`, and `(b c)` has two children `b` and `c`.

Another mechanism for passing flags to the frontend is the `-Xfrontend` flag. When this flag appears in a command-line invocation of the driver, the command line argument that comes immediately after is passed to the frontend:

```
$ swiftc a.swift b.swift -Xfrontend -dump-requirement-machine
```

The SIL intermediate form is described in [2].

2.1. Name Lookup

Name lookup is the process of resolving identifiers to declarations. The Swift compiler does not have a distinct "name binding" phase; instead, name lookup is queried from various points in the compilation process. Broadly speaking, there are two kinds of name lookup: *unqualified lookup* and *qualified lookup*. An unqualified lookup resolves a single identifier `foo`, while qualified lookup resolves an identifier `bar` relative to a base, such as `foo.bar`. There are also three important variations which are described immediately after the two fundamental kinds.

Unqualified lookup An unqualified lookup is always performed relative to the source location where the identifier actually appears. The source location may be inside of a primary file or secondary file.

The first time an unqualified lookup is performed inside a source file, a *scope tree* is constructed by walking the source file's abstract syntax tree. The root scope is the source file itself. Each scope has an associated source range, and zero or more child scopes; each child scope's source range must be a subrange of the source range of its parent, and the source ranges of sibling scopes are disjoint. Each scope introduces zero or more *variable bindings*.

Unqualified lookup first finds the innermost scope containing the source location, and proceeds to walk the scope tree up to the root, searching each parent node for bindings named by the given identifier. If the lookup reaches the root node, a *top-level lookup* is performed next. This will look for top-level declarations named by the given identifier, first in all source files of the current module, followed by all imported modules.

Qualified lookup A qualified lookup looks inside a list of type declarations for members with a given name. Starting from an initial list of type declarations, qualified lookup also visits the superclass of a class declaration, and conformed protocols.

The more primitive operation performed at each step is called a *direct lookup*, which searches inside a single type declaration and its extensions only, by consulting the type declaration's *lookup table*.

Module lookup A qualified lookup where the base is a module declaration searches for a top-level declaration in the given module and any other modules that it re-exports via `@_exported import`.

Dynamic lookup A qualified lookup where the base is the `AnyObject` type implements the legacy Objective-C behavior of a message send to `id`, which can invoke any method defined in any Objective-C class or protocol. In Swift, a dynamic lookup searches a global lookup table constructed from all `@objc` members of all classes and protocols. Any class can contain `@objc` members; the attribute can either be explicitly stated, or inferred if the method overrides an `@objc` method from the superclass. Protocol members are `@objc` only if the protocol itself is `@objc`.

Operator lookup Operator symbols are declared at the top level of a module. Operator symbols have a fixity (prefix, infix, or postfix), and infix operators also have a *precedence group*. Precedence groups are partially ordered with respect to other precedence groups. Standard operators like `+` and `*` and their precedence groups are thus defined in the standard library, rather than being built-in to the language itself.

An arithmetic expression like `2 + 3 * 6` is parsed as a *sequence expression*, which is a flat list of nodes and operator symbols. The parser does not know the precedence, fixity or associativity of the `+` and `*` operators. Indeed, it does not know that they exist at all. The *pre-check* phase of the expression type checker looks up operator symbols and transforms sequence expressions into the more familiar nested tree form.

Operator symbols do not themselves have an implementation; they are just names. An operator symbol can used as the name of a function implementing the operator on a specific type (for prefix and postfix operators) or a specific pair of types (for infix operators). Operator functions can be declared either at the top level, or as a member of a type. As far as a name lookup is concerned, the interesting thing about operator functions is that they are visible globally, even when declared inside of a type. Operator functions are found by consulting the operator lookup table, which contains top-level operator functions as well as member operator functions of all declared types.

When the compiler type checks the expression `2 + 3 * 6`, it must pick two specific operator functions for `+` and `*` among all the possibilities in order to make this expression type check. In this case, the overloads for `Int` are chosen, because `Int` is the default literal type for the literals `2`, `3` and `6`.

Listing 2.1 shows the definition of some custom operators and precedence groups. Note that the overload of `++` inside struct `Chicken` returns `Int`, and the overload of `++` inside struct `Sausage` returns `Bool`. The closure value stored in `fn` applies `++` to two anonymous closure parameters, `$0` and `$1`. While they do not have declared types, by simply coercing the *return type* to `Bool`, we are able to unambiguously pick the overload of `++` declared in `Sausage`. (Whether this is good style is an exercise for the reader.)

Listing 2.1.: Operator lookup in action

```
prefix operator <&>
infix operator ++: MyPrecedence
infix operator **: MyPrecedence

precedencegroup MyPrecedence {
  associativity: right
  higherThan: AdditionPrecedence
}

// Member operator examples
struct Chicken {
  static prefix func <&>(x: Chicken) {}
  static func ++(lhs: Chicken, rhs: Chicken) -> Int {}
}

struct Sausage {
  static func ++(lhs: Sausage, rhs: Sausage) -> Bool {}
}

// Top-level operator example
func **(lhs: Sausage, rhs: Sausage) -> Sausage {}

// Global operator lookup finds Sausage.++
// 'fn' has type (Sausage, Sausage) -> Bool
let fn = { ($0 ++ $1) as Bool }
```

Initially, infix operators defined their precedence as an integer value; Swift 3 introduced named precedence groups [3]. The global lookup for operator functions dates back to when all operator functions were declared at the top level. Swift 3 also introduced the ability to declare operator functions as members of types, but the global lookup behavior was retained [4].

2.2. Delayed Parsing

The above "compilation pipeline" model is a simplification of the actual state of affairs. Recall that in the case where the driver schedules multiple frontend jobs, the list of source files is partitioned into disjoint subsets, where each subset becomes the primary files of some frontend job. Ultimately, each frontend job only needs to generate machine code from the declarations in its primary files, so all stages from SILGen onward operate on the frontend job's primary files only.

However, the situation with parsing and type checking is more subtle. At a minimum, each frontend job must parse and type check its primary files. Furthermore, the partition of source files into frontend jobs is artificial and not visible to the user, and certainly a declaration in a primary file can reference declarations in secondary files. Therefore, in the general case, the abstract syntax tree for all secondary files must be available to a frontend job as well. On the other hand, it would be inefficient if every frontend job was required to fully parse all secondary files, because the time spent in the parser would be proportional to the number of frontend jobs multiplied by the number of source files, negating the benefits of parallelism.

The *delayed parsing* optimization solves this dilemma. When parsing a secondary file for the first time, syntax tree nodes for the bodies of top-level types, extensions and functions are not actually built. Instead, the parser operates in a high-speed mode where comments are skipped and pairs of braces are matched, but very little other work is performed. This constructs a "skeleton" representation of each secondary file. If the body of a type or extension in a secondary file is needed later—for example, because the type checking of a declaration in a primary file needs to perform a name lookup into this type—the source range of the declaration is parsed again, this time building the full syntax tree.

Operator lookup is incompatible with delayed parsing, because operator functions defined inside types are globally visible, as explained in the previous section. To deal with this, the parser looks for the keyword "`func`" followed by an operator symbol when skipping a type or extension body in a secondary file. The presence of this token sequence effectively disables delayed parsing for this declaration, because the first time an operator lookup is performed in the expression pre-checking pass, the bodies of all types containing operator functions are parsed again. Most types and extensions do not define operator functions, so this occurs rarely in practice.

The situation with `AnyObject` lookup is similar, since a method call on a value of type `AnyObject` must consult a global lookup table constructed from `@objc` members of classes, and the (implicitly `@objc`) members of `@objc` protocols. Unlike operator functions, classes and `@objc` protocols are quite common in Swift programs, so it would be unfortunate to penalize compile-time performance when `AnyObject` is a rarely-used feature. Instead, the solution is to eagerly parse classes and `@objc` protocols the first time a frontend job encounters a dynamic `AnyObject` method call.

There's actually one more complication here. Classes can be nested inside of other types, whose bodies are skipped if they appear in a secondary file. This is resolved with the same trick as operator lookup. When skipping the body of a type, the parser looks for occurrences of the "`class`" keyword. If the body contains this keyword, this type is parsed and its members visited recursively when building the `AnyObject` global lookup table.

Most Swift programs, even those making heavy use of Objective-C interoperability, do not contain a dynamic `AnyObject` method call in every source file, so delayed parsing remains effective.

Example 2.1. Listing 2.2 shows an example of this behavior. This program consists of three files. Suppose that the driver kicks off three frontend jobs, with a single primary file for each frontend job:

- The frontend job with the primary file `a.swift` will parse `b.swift` and `c.swift` as secondary files. The body of `g()` in `b.swift` is skipped, and the body of `Outer` in `c.swift` is skipped. The parser makes a note that `Outer` contains the `class` keyword. The function `f()` in `a.swift` contains a dynamic `AnyObject` method call, so this frontend job will construct the global lookup table, triggering parsing of `Outer` and `Inner` in `c.swift`.
- The frontend job with the primary file `b.swift` will parse `a.swift` and `c.swift` as secondary files. This primary file does not reference anything from `c.swift` at all, so `Outer` remains unparsed in this frontend job. Type checking the call to `f()` from `g()` also does not require parsing the *body* of `f()`.
- The frontend job with the primary file `c.swift` will parse `a.swift` and `b.swift` as secondary files, skipping parsing the bodies of `f()` and `g()`.

Example 2.2. It is possible to construct a program where type checking of each primary file triggers complete parsing of all type and extension bodies in every secondary file, either because of pathological dependencies between source files, or extreme reliance on operator lookup and `AnyObject` lookup. Listing 2.3 shows an example of the first kind. Again, if you assume the driver kicks off three frontend jobs with a single primary file for each frontend job, then each frontend job will eventually parse all type bodies in the other two secondary files.

Listing 2.2.: Delayed parsing with `AnyObject` lookup

```
// a.swift
func f(x: AnyObject) {
  x.foo()!
}
```

```
// b.swift
func g() {
  f()
}
```

```
// c.swift
struct Outer {
  class Inner {
    @objc func foo() {}
  }
}
```

2.3. Request Evaluator

The *request evaluator* is central to the architecture of the Swift compiler. Essentially, the request evaluator is a framework for performing queries against the abstract syntax tree. A *request* packages a list of input parameters together with an *evaluation function*. With the exception of emitting diagnostics, the evaluation function should be referentially transparent. Only the request evaluator should directly invoke the evaluation function; the request evaluator caches the result of the evaluation function for subsequent requests. As well as caching results, the request evaluator implements automatic cycle detection, and dependency tracking for incremental builds.

The request evaluator is used to implement a form of lazy type checking. We saw from the previous section that in any given frontend job, declarations in primary files can reference declarations in secondary files without restriction. Swift programmers also know that declarations in a source file can also appear in any order; there is no need to forward declare names, and certain kinds of circular references are also permitted.

For this reason the classic compiler design of a single type-checking pass that walks declarations in source order is not well-suited for Swift. Indeed, while the Swift type checker does walk over the declarations in each primary file over source order, instead of directly performing type checking work, it kicks off a series of requests which perform

Listing 2.3.: Defeating delayed parsing

```
// x.swift
struct A {
  typealias T = B.T
  typealias U = C.T
}
```

```
// y.swift
struct B {
  typealias T = C.T
  typealias U = A.T
}
```

```
// z.swift
struct C {
  typealias T = Int
  typealias U = B.T
}
```

Listing 2.4.: Forward reference example

```
1 let food = cook()
2 func cook() -> Food {}
3 struct Food {}
```

queries against declarations that may appear further down in the primary file, or in other secondary files. The compiler defines over two hundred kinds of requests. Important request kinds include:

- The **type-check source file request** is the key entry point into the type checker, explained below.
- The **AST lowering request** is the entry point into SILGen, generating SIL from the abstract syntax tree for a source file.
- The **unqualified lookup request** and **qualified lookup request** perform the two kinds of name lookup described in the previous section.
- The **interface type request** is explained in Chapter 4.
- The **generic signature request** is explained in Chapter 11.

The **type-check source file request**'s evaluation function visits each declaration in a primary source file. It is responsible for kicking off enough requests to ensure that SILGen can proceed if all requests succeeded without emitting diagnostics. Consider what happens when type checking the program in Listing 2.4:

1. The **type-check source file request** begins by visiting the declaration of `food` and performing various semantic checks.
2. One of these checks evaluates the **interface type request** with the declaration of `food`. This is a variable declaration, so the evaluation function type checks the initial value expression and returns the type of the result.
   a) In order to type check the expression `cook()`, the **interface type request** is evaluated again, this time with the declaration of `cook` as its input parameter.
   b) The interface type of `cook()` has not been computed yet, so the request evaluator calls the evaluation function for this request.
3. After computing the interface type of `food` and performing other semantic checks, the **type-check source file request** moves on to the declaration of `cook`:

Listing 2.5.: Diagnostic emitted during SILGen

```
// a.swift
struct Box {
  let contents: DoesNotExist
}
```

```
// b.swift
func open(_: Box) {}
```

a) The **interface type request** is evaluated once again, with the input parameter being the declaration of `cook`.

b) The result was already cached, so the request evaluator immediately returns the cached result without computing it again.

The **type-check source file request** is special, because it does not return a value; it is evaluated for the side effect of emitting diagnostics, whereas most other requests return a value. The implementation of the **type-check source file request** guarantees that if no diagnostics were emitted, then SILGen can generate valid SIL for all declarations in a primary file. However, SILGen can still evaluate other requests which result in diagnostics being emitted in secondary files.

Example 2.3. Listing 2.5 shows a program with two files. The first file declares a struct with a stored property naming a non-existent type. The second file declares a function whose input parameter type is the struct type declared by this struct declaration.

A frontend job with the primary file `b.swift` and the secondary file `a.swift` does not emit any diagnostics in the type checking pass, because the stored property `contents` of `Box` is not actually referenced.

However when SILGen runs, it needs to determine whether the parameter of type `Box` to the `open()` function needs to be passed directly in registers, or via an address by computing the *type lowering* for the `Box` type. Type lowering recursively visits the stored properties of `Box` and computes their type lowering; this evaluates the **interface type request** for the `contents` property of `Box`, which emits a diagnostic because the identifier "`DoesNotExist`" does not resolve to a valid type.

The request evaluator framework was first introduced in Swift 4.2 [5]. In subsequent releases, various ad-hoc mechanisms were gradually converted into request evaluator requests, with resulting gains to compiler performance, stability, and implementation maintainability.

Cycles In a language that supports forward references, it is possible to write a program that is syntactically well-formed, and where all identifiers resolve to valid declarations, but is nonetheless invalid because of circularity. The classic example of this is a pair of classes where each class inherits from the other:

```
class A: B {}
class B: A {}
```

Implementing bespoke logic to detect circularity is error-prone and tedious, and a missing circularity check can result in a crash or infinite loop when the compiler encounters an invalid input program. Instead, the request evaluator solves this problem in a more elegant way by maintaining a stack of *active requests*. When a request is evaluated, the request evaluator first checks if the active request stack contains a request with the same kind and equal input parameters. In this case, calling the evaluation function would result in infinite recursion, so instead the request evaluator diagnoses an error and returns a request-specific sentinel value. The circularity diagnostic can be customized for each request kind; the default just reports a "circular reference." If the compiler is invoked with the `-debug-cycles` frontend flag, the active request stack is also printed:

```
$ swiftc cycle.swift -Xfrontend -debug-cycles
===CYCLE DETECTED===
 `--TypeCheckSourceFileRequest(source_file "cycle.swift")
     `--SuperclassDeclRequest(cycle.(file).A@cycle.swift:1:7)
         `--SuperclassDeclRequest(cycle.(file).B@cycle.swift:2:7)
             `--SuperclassDeclRequest(cycle.(file).A@cycle.swift:1:7)
cycle.swift:1:7: error: 'A' inherits from itself
class A: B {}
      ^
cycle.swift:2:7: note: class 'B' declared here
class B: A {}
      ^
```

Debugging In addition to `-debug-cycles`, a couple of command-line flags help with debugging compile-time performance issues. The `-stats-output-dir` flag is followed by the name of a directory, which must already exist. Each frontend job writes a new JSON file to this directory, with various counters and timers. For each kind of request, there is a counter for the number of unique requests of this kind that were evaluated, not counting requests whose results were cached. The timer records the time spent in the request's evaluation function. The output can be sliced and diced in various ways; you can actually make pretty effective use of `awk`, despite the JSON format:

```
$ mkdir /tmp/stats
$ swiftc ... -stats-output-dir /tmp/stats
$ awk '/InterfaceTypeRequest.wall/ { x += $2 } END { print x }' \
    /tmp/stats/*.json
```

The second command-line flag is `-trace-stats-events`. It must be passed in conjunction with `-stats-output-dir`, and enables output of a trace file to the statistics directory. The trace file records a time-stamped event for the start and end of each request evaluation function, in CSV format.

2.4. Incremental Builds

The request evaluator also records dependencies for incremental compilation. The goal of incremental compilation is to prove which files do not need to be rebuilt, in the least conservative way possible. The quality of an incremental compilation implementation can be judged as follows:[2]

1. Perform a clean build of all source files in the program, and collect the object files.
2. Make a change to one or more source files in the input program.
3. Do an incremental build, which rebuilds some subset of source files in the input program. If a source file was rebuilt but the resulting object file is identical to the one saved in Step 1, the incremental build performed *wasted work*.
4. Finally, do another clean build, which yet again rebuilds all source files in the input program. If a source file was rebuilt and the resulting object file is different to the one saved in Step 1, the incremental build was *incorrect*.

This highlights the difficulty of the incremental compilation problem. Rebuilding too many files is an annoyance; rebuilding *too few* files is an error. A correct but ineffective implementation would rebuild all source files every time. The opposite approach of only rebuilding the subset of source files that have changed since the last compiler invocation is also too aggressive. To see why it is incorrect, consider the program shown in Listing 2.6. Let's say the programmer builds the program, adds the overload `f: (Int) -> ()`, then builds it again. The new overload is more specific, so the call `f(123)` in `b.swift` now refers to the new overload; therefore, `b.swift` must also be rebuilt.

The approach used by the Swift compiler is to construct a *dependency graph*. The frontend outputs a *dependency file* for each source file, recording all names the source file *provides*, and all names the type checker *requires* while compiling the source file.

[2] Credit for this idea goes to David Ungar.

Listing 2.6.: Rebuilding a file after adding a new overload

```
// a.swift
func f<T>(_: T) {}

// new overload added in second version of file
func f(_: Int) {}
```

```
// b.swift
func g() {
  f(123)
}
```

When performing an incremental build, the driver begins by rebuilding all source files which have changed since the last compilation, because at a minimum, these files need to be rebuilt. Then, the driver reads the dependency files, collecting all names provided by the changed source files, and rebuilds all source files which require those names.

Dependency files use a binary serialization format and have the "`.swiftdeps`" file name extension. The list of provided names in the dependency file is generated by walking the abstract syntax tree, collecting all visible declarations in each source file. The list of required names is generated by the request evaluator, using the stack of active requests. Every cached request has a list of required names, and a request can optionally be either a dependency sink, or dependency source.

A *dependency sink* is a name lookup request which records a required name. When a dependency sink request is evaluated, the request evaluator walks the stack of active requests, adding the identifier to each active request's list of required names. When a request with a cached value is evaluated again, the request's existing list of required names is "replayed," adding them to each active request that depends on the cached value.

A *dependency source* is a request which appears at the top of the request stack, such as the **type-check source file request** or the **AST lowering request**. After a dependency source request has been evaluated, its list of required names is added to the corresponding source file's list of required names.

Example 2.4. The above describes a subtle trick when evaluating a request whose result has already been cached. Listing 2.7 shows a program with three source files. Suppose now that the driver decides to compile *both* `a.swift` and `b.swift` in the same frontend job. This frontend job proceeds as follows:

1. First, the **type-check source file request** runs with the source file `a.swift`.

Listing 2.7.: Recording incremental dependencies

```
// a.swift
func breakfast() {
  soup(nil)
}
```

```
// b.swift
func lunch() {
  soup(nil)
}
```

```
// c.swift
func soup(_: Pumpkin?) {}
struct Pumpkin {}
```

a) While type checking the body of `breakfast()`, the type checker evaluates the **unqualified lookup request** with the identifier "`soup`."

b) This records the identifier "`soup`" in the requires list of each active request. There is one active request, the **type-check source file request** for `a.swift`.

c) The lookup finds the declaration of `soup()` in `c.swift`.

d) The type checker evaluates the **interface type request** with the declaration of `soup()`.

   i. The **interface type request** evaluates the **unqualified lookup request** with the identifier "`Pumpkin`."

   ii. This records the identifier "`Pumpkin`" in the requires list of each active request, of which there are now two: the **interface type request** for `soup()`, and the **type-check source file request** for `a.swift`.

e) The **type-check source file request** for `a.swift` has now finished. The requires list for this request contains two identifiers, "`soup`" and "`Pumpkin`"; both are added to the requires list of the source file `a.swift`.

2. Next, the **type-check source file request** runs with the source file `b.swift`.

   a) While type checking the body of `lunch()`, the type checker evaluates the **unqualified lookup request** with the identifier "`soup`."

   b) This records the identifier "`soup`" in the requires list of each active request. There is one active request, the **type-check source file request** for `b.swift`.

c) The lookup finds the declaration of `soup()` in `c.swift`.

d) The type checker evaluates the **interface type request** with the declaration of `soup()`.

e) This request has already been evaluated, and the cached result is returned. The requires list for this request is the single identifier "`Pumpkin`." This requires list is replayed, as if the request was being evaluated for the first time. This adds the identifier "`Pumpkin`" to the requires list of each active request, of which there is just one: the **type-check source file request** for `b.swift`.

f) The **type-check source file request** for `b.swift` has now finished. The requires list for this request contains two identifiers, "`soup`" and "`Pumpkin`"; both are added to the requires list of the source file `b.swift`.

Once this frontend job completes, dependency files for `a.swift` and `b.swift` are written out. Both source files require the names "`soup`" and "`Pumpkin`." The dependency of `b.swift` on "`Pumpkin`" is correctly recorded because evaluating a request with a cached value replays the request's requires list in Step (2.f) above.

There's a bit more to the story than this, but we're already far afield from the goal of describing Swift generics; you can find more details in [5] and [6].

2.5. Module System

The list of source files in a compiler invocation together form the *main module*. The main module is special, because its abstract syntax tree is constructed directly by parsing source code. There are three other kinds of modules: serialized modules, imported modules, and the built-in module.

A module is represented by a *module declaration* containing one or more *file units*. In the main module, the file units are *source files*, where each stores the parsed syntax tree for that source file. A serialized module contains one or more *serialized AST file units* and imported modules consist of one or more *Clang file units*.

The `import` keyword parses as an *import declaration*. After parsing, one of the first stages of type checking loads all modules imported by the main module. The standard library is defined in the `Swift` module, which is imported automatically unless the frontend was invoked with the `-parse-stdlib` flag, which is used when building the standard library itself.

As for the special `Builtin` module, it contains types and intrinsics implemented by the compiler itself, to be used when implementing the standard library. The `-parse-stdlib` flag also causes the built-in module to be implicitly imported (Section 3.4).

Serialized Modules A serialized module is output when the Swift compiler is invoked with the `-emit-module` flag. Serialized module files use the "`.swiftmodule`" file name extension. Serialized modules are stored in a binary format, closely tied to the specific version of the Swift compiler (when building a shared library for distribution, it is better to publish a textual interface instead, as described at the end of this section). Name lookup into a serialized module lazily constructs declarations by deserializing records from this binary format as needed.

Deserialized declarations generally look like parsed declarations that have already been type checked, but they sometimes contain less information. For example, in Chapter 5, you will see various syntactic representations of generic parameter lists, `where` clauses, and so on. Since this information is only used when type checking the declaration, it is not serialized. Instead, deserialized declarations only need to store a generic signature, described in Chapter 6.

Another key difference between parsed declarations and deserialized declarations is that parsed function declarations have a body, consisting of statements and expressions. This body is never serialized, so deserialized function declarations never have a body. The one case where the body of a function is made available across module boundaries is when the function is annotated with the `@inlinable` attribute; this is implemented by serializing the SIL representation of the function instead.

Imported Modules An imported module is implemented in C, Objective-C or C++. The Swift compiler embeds a copy of Clang and uses it to parse module maps, header files, and binary precompiled headers. Name lookup into an imported module lazily constructs Swift declarations from their corresponding Clang declarations. The Swift compiler component responsible for this is known as the "ClangImporter."

Imported function declarations generally do not have bodies if the entry point was previously emitted by Clang and is available externally. Occasionally the ClangImporter synthesizes accessor methods and other such trivia, which do have bodies represented as Swift statements and expressions. C functions not available externally, such as `static inline` functions declared in header files, are emitted by having Swift IRGen call into Clang.

Invoking the compiler with the `-import-objc-header` flag followed by a header file name specifies a *bridging header*. This is a shortcut for making C declarations in the bridging header visible to all other source files in the main module, without having to define a separate Clang module first. This is implemented by adding a Clang file unit corresponding to the bridging header to the main module. For this reason, you should not assume that all file units in the main module are necessarily source files.

Textual Interfaces The Swift binary module format depends on compiler internals and no attempt is made to preserve compatibility across compiler releases. When building a

shared library for distribution, you can instead generate a *textual interface*:

```
$ swiftc Horse.swift -enable-library-evolution -emit-module-interface
```

The `-enable-library-evolution` flag enables *resilience*, which instructs client code to use more abstract access patterns which are guaranteed to only depend on the published public declarations of a module. For example, this allows adding new fields to a public struct, since client code is required to pass the struct indirectly. Library evolution is a prerequisite for emitting a textual interface; unlike the serialized module format, textual interfaces only describe the public declarations of a module.

Textual interface files use the "`.swiftinterface`" file name extension. They are generated by the AST printer, which prints declarations in a format that looks very much like Swift source code, with a few exceptions:

1. Non-`@inlinable` function bodies are skipped. Bodies of `@inlinable` functions are printed verbatim, including comments, except that `#if` conditions are evaluated.
2. Various synthesized declarations, such as type alias declarations from associated type inference, witnesses for derived conformances such as `Equatable`, and so on, are written out explicitly.
3. Opaque return types also require special handling (Section 14.2).

Note that (1) above means the textual interface format is target-specific; a separate textual interface needs to be generated for each target platform, alongside the shared library itself.

When a module defined by a textual interface is imported for the first time, a frontend job parses and type checks the textual interface, and generates a serialized module file which is then consumed by the original frontend job. Serialized module files generated in this manner are cached, and can be reused between invocations of the same compiler version.

The `@inlinable` attribute was introduced in Swift 4.2 [7]. The Swift ABI was formally stabilized in Swift 5.0, when the standard library became part of the operating system on Apple platforms. Library evolution support and textual interfaces became user-visible features in Swift 5.1 [8].

2.6. Source Code Reference

The Swift driver is now implemented in Swift, and lives in a separate repository from the rest of the compiler:

https://github.com/apple/swift-driver

The Swift frontend, standard library and runtime are found in the main repository:

https://github.com/apple/swift

The major components of the Swift frontend live in their own subdirectories of the main repository. The entities modeling the abstract syntax tree are defined in lib/AST/ and include/swift/AST/; among these, types and declarations are important for the purposes of this book, and will be covered in Chapter 3 and Chapter 4. The core of the SIL intermediate language is implemented in lib/SIL/ and include/swift/SIL/.

Each stage of the compilation pipeline has its own subdirectory:

- lib/Parse/
- lib/Sema/
- lib/SILGen/
- lib/SILOptimizer/
- lib/IRGen/

The AST Context

Key source files:

- include/swift/AST/ASTContext.h
- lib/AST/ASTContext.cpp

ASTContext	*class*

The global singleton for a single frontend instance. An AST context provides a memory allocation arena, unique allocation for various immutable data types used throughout the compiler, and storage for various other global singletons.

Request Evaluator

Key source files:

- include/swift/AST/Evaluator.h
- lib/AST/Evaluator.cpp

`SimpleRequest` *template class*

Each request kind is a subclass of `SimpleRequest`. The evaluation function is implemented by overriding the `evaluate()` method of `SimpleRequest`.

`RequestFlags` *enum class*

One of the template parameters to `SimpleRequest` is a set of flags:

- `RequestFlags::Uncached`: indicates that the result of the evaluation function should not be cached.
- `RequestFlags::Cached`: indicates that the result of the evaluation function should be cached by the request evaluator, which uses a per-request kind `DenseMap` for this purpose.
- `RequestFlags::SeparatelyCached`: the result of the evaluation function should be cached by the request implementation itself, as described below.
- `RequestFlags::DependencySource`, `DependencySink`: if one of these is set, the request kind becomes a dependency source or sink, as described in Section 2.4.

Separate caching can be more performant if it allows the cached value to be stored directly inside of an AST node, instead of requiring the request evaluator to consult a side table. For example, many requests taking a declaration as input store the result directly inside of the `Decl` instance or some subclass thereof.

Due to expressivity limitations in C++, a bit of boilerplate is involved in the definition of a new request kind. For example, consider the `InterfaceTypeRequest`, which takes a `ValueDecl` as input and returns a `Type` as output:

- The request type ID is declared in `include/swift/AST/TypeCheckerTypeIDZone.def`.
- The `InterfaceTypeRequest` class is declared in `include/swift/AST/TypeCheckRequests.h`.
- The `InterfaceTypeRequest::evaluate()` method is defined in `lib/Sema/TypeCheckDecl.cpp`.
- The request is separately cached. The `InterfaceTypeRequest` class overrides the `isCached()`, `getCachedResult()` and `cacheResult()` methods to store the declaration's interface type inside the `ValueDecl` instance itself. These methods are implemented in `lib/AST/TypeCheckRequestFunctions.cpp`.

`Evaluator` *class*

Request evaluation is performed by calling the `evaluateOrDefault()` top-level function, passing it an instance of the request evaluator, the request to evaluate, and a sentinel value to return in case of circularity. The `Evaluator` class is a singleton, stored in the `evaluator` instance variable of the global `ASTContext` singleton. The request evaluator will either return a cached value, or invoke the evaluation function and cache the result. For example, the `getInterfaceType()` method of `ValueDecl` is implemented as follows:

```
Type ValueDecl::getInterfaceType() const {
  auto &ctx = getASTContext();
  return evaluateOrDefault(
    ctx.evaluator,
    InterfaceTypeRequest{const_cast<ValueDecl *>(this)},
    ErrorType::get(ctx)));
}
```

Name Lookup

Key source files:

- include/swift/AST/NameLookup.h
- include/swift/AST/NameLookupRequests.h
- lib/AST/NameLookup.cpp
- lib/AST/UnqualifiedLookup.cpp

The "AST scope" subsystem implements unqualified lookup for local bindings. Outside of the name lookup implementation itself, the rest of the compiler does not generally interact with it directly:

- include/swift/AST/ASTScope.h
- lib/AST/ASTScope.cpp
- lib/AST/ASTScopeCreation.cpp
- lib/AST/ASTScopeLookup.cpp
- lib/AST/ASTScopePrinting.cpp
- lib/AST/ASTScopeSourceRange.cpp

`UnqualifiedLookupRequest` *class*

Unqualified lookups are performed by evaluating an instance of this request kind. The request takes an `UnqualifiedLookupDescriptor` as input.

`UnqualifiedLookupDescriptor` *class*

Encapsulates the input parameters for an unqualified lookup:

- The name to look up.
- The declaration context where the lookup starts.
- The source location where the name was written in source. If not specified, this becomes a top-level lookup.
- Various flags, described below.

`UnqualifiedLookupFlags` *enum class*

Flags passed as part of an `UnqualifiedLookupDescriptor`.

- `UnqualifiedLookupFlags::TypeLookup`: if set, lookup ignores declarations other than type declarations. This is used in type resolution.
- `UnqualifiedLookupFlags::AllowProtocolMembers`: if set, lookup finds members of protocols and protocol extensions. Generally should always be set, except to avoid request cycles in cases where it is known the result of the lookup cannot appear in a protocol or protocol extensions.
- `UnqualifiedLookupFlags::IgnoreAccessControl` if set, lookup ignores access control. Generally should never be set, except when recovering from errors in diagnostics.
- `UnqualifiedLookupFlags::IncludeOuterResults` if set, lookup stops after finding results in an innermost scope, or to always proceed to a top-level lookup.

`DeclContext` *class*

Declaration contexts will be introduced in Chapter 4, and the `DeclContext` class in Section 4.4.

- `lookupQualified()` has various overloads, which perform a qualified name lookup into one of various combinations of types or declarations. The "`this`" parameter—the `DeclContext *` which the method is called on determines the visibility of declarations found via lookup through imports and access control; it is not the base type of the lookup.

`NLOptions` *enum*

Similar to `UnqualifiedLookupFlags`, but for `DeclContext::lookupQualified()`.

- `NL_OnlyTypes`: if set, lookup ignores declarations other than type declarations. This is used in type resolution.
- `NL_ProtocolMembers`: if set, lookup finds members of protocols and protocol extensions. Generally should always be set, except to avoid request cycles in cases where it is known the result of the lookup cannot appear in a protocol or protocol extension.
- `NL_IgnoreAccessControl`: if set, lookup ignores access control. Generally should never be set, except when recovering from errors in diagnostics.

`NominalTypeDecl` *class*

Nominal type declarations will be introduced in Chapter 4, and the `NominalTypeDecl` class in Section 4.4.

- `lookupDirect()` performs a direct lookup, which only searches the nominal type declaration itself, and its extensions.

Primary File Type Checking

Key source files:

- `lib/Sema/TypeCheckDeclPrimary.cpp`

The `TypeCheckSourceFileRequest` calls the `typeCheckDecl()` global function, which uses the visitor pattern to switch on the declaration kind. For each declaration kind, it performs various semantic checks and kicks off requests which may emit diagnostics.

Module System

`ModuleDecl` *class*

A module.

- `getName()` returns the module's name.
- `getFiles()` returns an array of `FileUnit`.
- `isMainModule()` answers if this is the main module.

`FileUnit` *class*

Abstract base class representing a file unit.

`SourceFile` *class*

Represents a parsed source file from disk. Inherits from `FileUnit`.

- `getTopLevelDecls()` returns an array of all top-level declarations in this source file.
- `isPrimary()` returns `true` if this is a primary file, `false` if this is a secondary file.
- `isScriptMode()` answers if this is the main file of a module.
- `getScope()` returns the root of the scope tree for unqualified lookup.

Imported and serialized modules get a subdirectory each:

- `lib/ClangImporter/`
- `lib/Serialization/`

The AST printer for generating textual interfaces is implemented in a pair of files:

- `include/swift/AST/ASTPrinter.h`
- `lib/AST/ASTPrinter.cpp`

The interface between name lookup and the module system is mediated by a pair of abstract base classes defined in the below header file:

- `include/swift/AST/LazyResolver.h`

`LazyMemberLoader` *class*

Abstract base class implemented by different kinds of modules to look up top-level declarations and members of types and extensions. For the main module, this consults lookup tables, for serialized modules this deserializes records and builds declarations from them, for imported modules this constructs Swift declarations from Clang declarations.

`LazyConformanceLoader` *class*

Abstract base class implemented by different kinds of modules to fill out conformances (Chapter 8).

3. Types

Swift makes a distinction between a *type representation*, read by the parser, and a *type*, which is a semantic object understood by the type checker. Type representations are resolved to types by performing *type resolution*.

Not all types are constructed by resolving type representations written in source; building types and taking them apart is an extremely common activity throughout the compiler.

Notation A few words about the notation used throughout this book. For convenience, we identify the printed form of a type, such as `Array<Int>`, with its type representation and the semantic type, depending on context. But what does it actually mean to say that the string "`Array<Int>`" parses into the type representation `Array<Int>` which resolves into the type `Array<Int>`?

First, we have the string "`Array<Int>`" written somewhere in a source file. The lexer splits the string up into a sequence of tokens: "`Array`", "`<`", "`Int`", and "`>`". The parser reads each token, building up a type representation.

A type representation has a tree structure, so when we talk about the type representation `Array<Int>`, we really mean this:

> "An identifier type representation with a single component, storing the identifier `Array` together with a single generic argument. The generic argument is another identifier type representation, again with a single component, storing the identifier `Int`."

Types also have a tree structure, so when we talk about the type `Array<Int>`, what we really mean is:

> "A generic nominal type for the struct declaration named `Array`, with a single generic argument. The single generic argument is a nominal type for the struct declaration named `Int`."

The difference between the type representation `Array<Int>` and the type `Array<Int>` is that the type representation only stores identifiers, with no connection to the declaration of `Array` and `Int`. The semantic type points at the declarations themselves.

There is also a notion of validity here. The string "`Foo>(<Bar`" is neither a type representation nor a type. The strings `Array<Int, String>` and `Pasta<TomatoSauce>`

can both be seen as type representations, but the former does not resolve to a valid type because `Array` only has a single generic argument. The latter only resolves to a valid type if a type declaration named `Pasta` exists, and if `TomatoSauce` also resolves to a valid type.

Type representations are rarely encountered outside of the type resolution process and the parser itself, so we will leave them aside until Chapter 10.

Tree structure Types are categorized into kinds, such as nominal types, function types, and so on. Each kind of type is composed of smaller structural components, including other types, pointers to declarations, and various attributes. Factory methods for each kind construct types from their structural components, and analogously, each kind has getter methods to take the type apart. Once created, types are immutable.

This gives types a recursive tree structure. To say that a type *contains* another type means that the latter appears as a child node of the former. This concept is most useful when talking about types containing generic parameters, because those types can be substituted to form concrete types. For example, if `T` is a generic parameter type, the type `Array<T>` can be substituted by replacing `T` with `Int`. This gives you the type `Array<Int>`, which no longer contains any generic parameters.

Various utility operations exist to walk the recursive structure of a type, check if it contains a type with certain properties, or transform its contained types, forming a new type with the same tree structure.

Canonical types The Swift grammar defines some shorthand spellings for common types, such as `[T]` for `Array<T>` and `T?` for `Optional<T>`. Type alias declarations are another kind of shorthand; declaring a type alias type introduces a new name for some existing type. The various alternate spellings for existing types are called *sugared types*; Section 3.3 gives a full account of the possible kinds.

A type is *canonical* if it does not contain any sugared types. Computing the canonical type of an arbitrary type returns the original type if it was already canonical, otherwise it transforms the type by replacing all sugared types that it contains with their desugared form.

The compiler tries to preserve type sugar when resolving type representations into types and when transforming types, ensuring that types mentioned in diagnostics look like the types written by the user. After type checking, compiler passes such as SILGen and IRGen mostly deal with canonical types.

For the most part, type sugar has no semantic effect. For example, it would not make sense to define two overloads of the same function that only differ by sugared types. One notable exception is the rule for default initialization of variables: if the variable's type is declared as the sugared optional type `T?` for some `T`, the variable's initial value expression is assumed to be `nil` if none was provided. Spelling the type as `Optional<T>`

Listing 3.1.: The sugared optional type has a semantic effect

```
var x: Int?
print(x)  // prints 'nil'

var y: Optional<Int>
print(y)  // error: use of uninitialized variable 'y'
```

avoids the default initialization behavior. Listing 3.1 shows an example of this rule.

Type equality In addition to being immutable, types are uniquely allocated; if the type `Array<Int>` is constructed twice in the same compilation instance, both values will point at the same type object in memory. Three levels of equality are defined on types, from strongest to weakest:

1. **Pointer equality** determines if two types are exactly equal as trees. The type `Array<Int>` is not pointer-equal to the sugared type `[Int]`.
2. **Canonical equality** determines if two types have the same canonical type. The types `Array<Int>` and `[Int]` are canonical-equal, because the canonical type of `[Int]` is `Array<Int>`.
3. **Reduced equality** determines if two types have the same reduced type with respect to a generic signature. Two different type parameters can reduce to the same type parameter when same-type requirements are in play. Reduced types are formally introduced in Section 6.4.

If both types are already canonical, the first two relations coincide; if both types are reduced, all three coincide.

The remainder of this chapter describes, for each kind of type, the role it plays in the language and how it breaks down into structural components.

Nominal types A *nominal type* is the type declared by a non-generic struct, enum or class declaration, such as `Int`. A *generic nominal type* is a type declared by a generic struct, enum or class declaration, *specialized* with a list of generic arguments, such as `Array<Int>`.

Both kinds of nominal types point at their declaration. They also store a parent type if the nominal type declaration is nested inside of another nominal type declaration. The parent type can be a sugared type, but its canonical type must always be the correct nominal type for the original type declaration's parent type declaration. Nested type declarations are described in Section 4.1.

3.1. Structural Types

Structural types are those built-in to the language, rather than being defined in the standard library or user code. Structural types are not to be confused with the types produced by the *structural resolution stage*, which is discussed in Chapter 10.

Tuple types A tuple type is an ordered list of element types with optional labels. A value of a tuple type is a list of values with the corresponding element types. The list of element types can be empty, which gives the unique empty tuple type `()`. The standard library declares a type alias `Void` whose underlying type is `()`.

From the user's point of view, tuple types are either empty or have at least two elements. An unlabeled one-element tuple type cannot be formed at all; `(T)` resolves to the same type as `T` in the language.

Labeled one-element tuple types have a production in the grammar, but are explicitly rejected by type resolution. They can still appear in the implementation when SILGen needs to materialize the associated value of an enum case as a single value (for instance, `case person(name: String)`), but such types cannot arise as the types of expressions nor can they be written in source.

Function types The type of a function declaration or closure expression is a function type. In the expression grammar, *call expressions* are formed from an expression with a function type, such as a declaration reference, closure expression or result of another call, together with an argument list.

A function type stores a parameter list, a return type, and attributes. The attributes includes the function's effect, `@escaping` attribute, and an optional calling convention:

- The two effect kinds are `throws` and `async`; the latter was introduced as part of the concurrency model in Swift 5.5 [9].
- Non-escaping functions are second-class and can only be passed to other functions, captured by non-escaping closures, or immediately called; they cannot be stored inside other values.
- A `@convention(thin)` function is passed as a single function pointer, without a closure context; it cannot capture values from outer scopes.
- A `@convention(c)` function is similarly restricted and also must have parameter and return types representable in C.
- A `@convention(block)` function is an Objective-C block, which allow captures but must have parameter and return types representable in Objective-C.

Listing 3.2.: "Tuple splat" function conversion example

```
1 func apply<T, U>(fn: (T) -> U, arg: T) -> U {
2   return fn(arg)
3 }
4
5 print(apply(fn: (+), arg: (1, 2)))  // prints 3
```

Each entry in the parameter list consists of a parameter type and again, some non-type bits:

- the **value ownership kind**, which is one of default, `inout`, `__owned` or `__shared`,
- the **variadic** flag, in which case the parameter type must be an array type.
- the `@autoclosure` attribute, in which case the parameter type must be another function type of the form `() -> T` for some type `T`.

When type checking a call to function value with a variadic parameter, the type checker collects multiple expressions from the call argument list into an implicit array expression. Otherwise, variadic parameters behave exactly like arrays once you get to SILGen and below.

The `@autoclosure` attribute instructs the type checker to treat the corresponding argument in the caller as if it were a value of type `T`, rather than a function type `() -> T`. The argument is then wrapped inside an implicit closure expression. In the body of the callee, an `@autoclosure` parameter behaves exactly like an ordinary function value, and can be called to evaluate the expression provided by the caller.

Note that the following are two different function types:

```
(Int, Int) -> Bool
((Int, Int)) -> Bool
```

The first has a parameter list with two entries, both storing the parameter type `Int`. The second has a parameter list with a single entry, storing a tuple type of two elements, `(Int, Int)`. The type checker does define an implicit conversion between them though, in the special case of passing a call argument. This allows the code in Listing 3.2 to type check.

Listing 3.3 demonstrates another subtle point. Argument labels are part of a function declaration's *name*, not a function declaration's *type*. A closure is always called without argument labels. This includes the case of a closure formed from an unapplied reference to a function declaration—even if the function declaration has argument labels.

Listing 3.3.: Argument labels are not part of a function declaration's type

```
func subtract(minuend x: Int, subtrahend y: Int) -> Int {
  return x - y
}

print(subtract(minuend: 3, subtrahend: 1))  // prints 2

let fn1 = subtract  // declaration name can omit argument labels
print(fn1(3, 1))  // prints 2

let fn2 = subtract(minuend:subtrahend:)  // full declaration name
print(fn2(3, 1))  // prints 2
```

The history of Swift function types is an interesting case study in language evolution. Originally, a function type always had a *single* input type, which could be a tuple type to model a function of multiple arguments. Tuple types used to be able to represent `inout` and variadic elements, and furthermore, the argument labels of a function declaration were part of the function declaration's type. The existence of such "non-materializable" tuple types introduced complications throughout the type system, and argument labels had inconsistent behavior in different contexts.

The syntax for referencing a declaration name with argument labels was adopted in Swift 2.2 [10]. Subsequently, argument labels were dropped from function types in Swift 3 [11]. The distinction between a function taking multiple arguments and a function taking a single tuple argument was first hinted at in Swift 3 with [12] and [13], and became explicit in Swift 4 [14]. At the same time, Swift 4 also introduced the "tuple splat" function conversion which simulated the Swift 3 model in a limited way for the cases where the old behavior was convenient. For example, the element type of `Dictionary` is a key/value tuple, but you often want to call the `Collection.map()` method with a closure taking two arguments, and not a closure with a single tuple argument.

Even after the above proposals were implemented, the compiler continued to model a function type as having a single input type for quite some time, despite this being completely hidden from the user. After Swift 5, the function type representation fully converged with the semantic model of the language.

Generic function types A generic function type is a function type adorned with a generic signature. Generic function types only appear as the interface type of a function declaration in a generic context.

Swift's type system does not support rank-2 polymorphism, so an expression in the Swift language can never have a generic function type. When referenced from within an expression, the interface type of a generic function declaration always has substitutions applied, making the type of the expression into a non-generic function type.

Generic function types have a special behavior when their canonical type is computed. Since generic function types carry a generic signature, the parameter types and return type of a *canonical* generic function type are actually *reduced* types with respect to this generic signature (Section 6.3).

Metatype types Types are values in Swift, and a metatype is the type of a type used as a value. The metatype of a type `T` is written as `T.Type`. The type `T` is the *instance type* of the metatype. For example `(() -> ()).Type` is the metatype type with the instance type `() -> ()`. This metatype has one value, the function type `() -> ()`.

Metatypes are sometimes referred to as *concrete metatypes*, to distinguish them from *existential metatypes*. Most concrete metatypes are singleton types, where the only value is the instance type itself. One exception is class metatypes for non-final classes; the values of a class metatype include the class type itself, but also all subclasses of the class.

3.2. Abstract Types

Generic parameter types A generic parameter type is the declared interface type of a generic parameter declaration. The sugared form references the declaration, and the canonical form only stores a depth and an index. This is all described in Chapter 5.

Care must be taken not to print canonical generic parameter types in diagnostics, to avoid surfacing the "`τ_1_2`" notation to the user. Section 6.5 shows a trick to transform a canonical generic parameter type back into its sugared form using a generic signature.

Dependent member types A dependent member type stores a base type together with an identifier or an associated type declaration. The former is called an *unbound* dependent member type, and the latter is *bound*. Unbound and bound dependent member types do not present as different concepts in the language.

Instead, Chapter 10 describes how dependent member types written in source can be first resolved from a type representation into their unbound form, and then resolved again into a bound form once a generic signature is available. We will write `T.A` for the unbound dependent member type with base type `T` and identifier `A`, or `T.[P]A` for the bound dependent member type with base type `T` and associated type declaration `A` from protocol `P`. The latter is not valid Swift syntax, but the notation is useful to distinguish the two.

A dependent member type is *proper* if the base type is a generic parameter or another proper dependent member type. Improper dependent member types appear internally in

the expression type checker and associated type inference, but are not ever constructed by type resolution. For the most part you can ignore them.

A *type parameter* is a generic parameter type or a proper dependent member type. A type that might contain type parameters but is not necessarily a type parameter itself is called an *interface type*.

Type parameters are discussed in Section 6.2 and Section 6.3.

Archetype types Type parameters only have meaning when considered together with a generic signature; archetypes are an alternate "self-describing" representation that stores their local requirements. An archetype is always part of a *generic environment*, a concept introduced in Chapter 9.

Archetypes store a reduced type parameter together with their generic environment, and the generic environment stores its generic signature; this signature describes the requirements imposed on the archetype's type parameter.

In the source language, archetypes and type parameters do not present as distinct concepts, but the compiler uses them internally in different contexts. In diagnostics, an archetype is printed as the type parameter it represents. To distinguish an archetype from its type parameter, we're going to use the notation ⟦T⟧.

Archetypes are also used to represent opaque return types (Section 14.1) and opened existential types (Section 15.1).

A type that might contain archetypes but is not necessarily an archetype itself is called a *contextual type*.

Protocol types A protocol type is the declared interface type of a protocol declaration. Protocol types are nominal types, but they never have generic arguments or parent types, and so there is exactly one protocol type for a given protocol declaration.

A protocol type is also a special kind of type called a *constraint type*, described in Section 5.1. A protocol type represents a conformance requirement to its protocol. A protocol type is never the type of a value in Swift; the concept of a type-erased container is represented with an existential type.

Protocol composition types A protocol composition type is a constraint type with a list of members. On the right hand side of a conformance requirement, protocol compositions *expand* into a list of generic requirements, one for each member of the composition, as described in Section 11.2. The members can include protocol types, a class type (at most one), and the `AnyObject` layout constraint:

```
P & Q
P & AnyObject
SomeClass<Int> & P
```

Parameterized protocol types A parameterized protocol type[1] stores a protocol type together with a list of generic arguments. As a constraint type, it expands to a conformance requirement together with one or more same-type requirements. The same-type requirements constrain the protocol's *primary associated types*, which are declared with a syntax similar to a generic parameter list.

The written representation looks like a generic nominal type, except the named declaration is a protocol, for example `Sequence<Int>`. Full details appear in Section 5.4. Parameterized protocol types were introduced in Swift 5.7 [15].

Existential types An existential type wraps a constraint type and represents a value with some unknown dynamic type satisfying this constraint. The written syntax is `any P`, where `P` is a constraint type. The `any` keyword was introduced in Swift 5.6 [16]. Prior to Swift 5.6, existential types and constraint types were the same concept, both in the language and in the compiler implementation. The existential type wrapper was introduced at the same time as the `any` keyword. Existential types are described in Chapter 15.

Existential metatype types An existential metatype represents a metatype value whose instance type satisfies some constraint type. For example, if `P` is a protocol type, the values of the existential metatype `any P.Type` are the concrete metatypes of types conforming to `P`.

An existential metatype is distinct from the *concrete* metatype of an existential type, which is the type with one value, the existential type `any P`.

Before the introduction of the `any` keyword, existential metatypes were written as `P.Type`, and the concrete metatype of an existential as `P.Protocol`. This created an edge case, because for all non-protocol types `T`, `T.Type` is always the concrete metatype.

The new spelling for an existential metatype is `any P.Type` or `any (P.Type)`, while the concrete metatype for the existential type itself is now written as `(any P).Type`—note that the parentheses follow the tree structure of the types.

Dynamic Self types The dynamic Self type appears when a class method declares a return type of `Self`. In this case, the object is known to have the same dynamic type as the base of the method call, which might be a subclass of the method's class. The dynamic Self type wraps a class type, which is the static upper bound for the type.

This concept comes from Objective-C, where it is called `instancetype`. The dynamic Self type in many ways behaves like a generic parameter, but it is not represented as one; the type checker and SILGen implement support for it directly.

[1] The evolution proposal calls them "constrained protocol types"; here we're going to use the terminology that appeared in the compiler implementation itself. Perhaps the latter should be renamed to match the evolution proposal at some point.

Listing 3.4.: Dynamic Self type example

```
class Base {
  required init() {}

  func dynamicSelf() -> Self {
    // the type of 'self' in a method returning 'Self' is
    // the dynamic Self type.
    return self
  }

  func clone() -> Self {
    return Self()
  }

  func invalid1() -> Self {
    return Base()
  }

  func invalid2(_: Self) {}
}

class Derived: Base {}

let y = Derived().dynamicSelf()  // y has type 'Derived'
let z = Derived().clone()  // z has type 'Derived'
```

Example 3.1. Listing 3.4 demonstrates some of the behaviors of the dynamic Self type. Two invalid cases are shown; `invalid1()` is rejected because the type checker cannot prove that the return type is always an instance of the dynamic type of `self`, and `invalid2()` is rejected because `Self` appears in contravariant position.

Note that `Self` has a different interpretation inside a non-class type declaration. In a protocol declaration, `Self` is the implicit generic parameter (Section 5.4). In a struct or enum declaration, `Self` is the declared interface type (Section 10.1).

3.3. Sugared Types

Sugared generic parameter types were already described in the previous section. Of the remaining kinds of sugared types, type alias types are defined by the user, and the other three are built-in to the language.

Type alias types A type alias type represents a reference to a type alias declaration. It contains an optional parent type, a substitution map, and the substituted underlying type. The canonical type of a type alias type is the substituted underlying type.

The type alias type's substitution map is formed in type resolution, from any generic arguments applied to the type alias type declaration itself, together with the generic arguments of the base type (Section 10.1). Type resolution applies this substitution map to the underlying type of the type alias declaration to compute the substituted underlying type. The type alias type also preserves this substitution map for printing, and for requirement inference (Section 11.1).

Optional types The optional type is written as `T?` for some object type `T`; its canonical type is `Optional<T>`.

Array types The array type is written as `[E]` for some element type `E`; its canonical type is `Array<E>`.

Dictionary types The dictionary type is written as `[K: V]` for some key type `K` and value type `V`; its canonical type is `Dictionary<K, V>`.

3.4. Built-in Types

What users think of as fundamental types, such as `Int` and `Bool`, are defined as structs in the standard library. These structs wrap the various *built-in types* which are understood directly by the compiler.

Built-in types are not nominal types, so they cannot contain members, cannot have new members added via extensions, and cannot conform to protocols. Values of built-in

types are manipulated using special *compiler intrinsics*. The standard library wraps built-in types in nominal types, and defines methods and operators on those nominal types which call the intrinsic functions, thereby presenting the actual interface expected by users.

For example, the `Int` struct defines a single stored property named `_value` with type `Builtin.Int`. The `+` operator on `Int` is implemented by extracting the `_value` stored property from a pair of `Int` values, calling the `Builtin.sadd_with_overflow_Int64` compiler intrinsic to add them together, and finally, wrapping the resulting `Builtin.Int` in a new instance of `Int`.

Built-in types and their intrinsics are defined in the special `Builtin` module, which is a special module constructed by the compiler itself and not built from source code. The `Builtin` module is only visible when the compiler is invoked with the `-parse-stdlib` frontend flag; the standard library is built with this flag, but user code never interacts with the `Builtin` module directly.

3.5. Miscellaneous Types

A handful of special types do not describe the types of values, but are used by the type checker as part of the type checking process.

Reference storage types A reference storage type is the type of a variable declaration adorned with the `weak`, `unowned` or `unowned(unsafe)` attribute. The wrapped type must be a class type, a class-constrained archetype, or class-constrained existential type. Reference storage types arise as the interface types of variable declarations, and as the types of SIL instructions. The types of expressions never contain reference storage types.

Placeholder types A placeholder type represents a generic argument to be inferred by the type checker. The written representation is the underscore "_". They can only appear in a handful of restricted contexts and do not remain after type checking. The expression type checker replaces placeholder types with type variables, solves the constraint system, and finally replaces the type variables with their fixed concrete types. For example, here the interface type of the `myPets` local variable is inferred as `Array<String>`:

```
1 let myPets: Array<_> = ["Zelda", "Giblet"]
```

Placeholder types were introduced in Swift 5.6 [17].

Unbound generic types Unbound generic types predate placeholder types, and can be seen as a special case. An unbound generic type is written as a named reference to a generic type declaration, without generic arguments applied. An unbound generic type

behaves like a generic nominal type where all generic arguments are placeholder types. In the example above, the generic nominal type `Array<_>` contains a placeholder type. The unbound generic type `Array` could have been used instead:

```
let myPets: Array = ["Zelda", "Giblet"]
```

One other place where unbound generic types can appear is in the underlying type of a non-generic type alias, which is shorthand for declaring a generic type alias that forwards its generic arguments. For example, the following declarations two are equivalent:

```
typealias MyDictionary = Dictionary
typealias MyDictionary<Key, Value> = Dictionary<Key, Value>
```

Unbound generic types are also occasionally useful in diagnostics when you want to print the name of a type declaration only (like `Outer.Inner`) without the generic parameters of its declared interface type (`Outer<T>.Inner<U>` for example).

Type variable types A type variable represents the future inferred type of an expression in the expression type checker's constraint system. The expression type checker builds the constraint system by walking an expression recursively, assigning new type variables to the types of sub-expressions and recording constraints between these type variables. Solving the constraint system can have three possible outcomes:

- **One solution**—every type variable has exactly one fixed type assignment; the expression is well-typed.
- **No solutions**—some constraints could not be solved, indicating erroneous input.
- **Multiple solutions**—the constraint system is underspecified and some type variables can have multiple valid fixed type assignments.

In the case of multiple solutions, the type checker uses heuristics to pick the "best" solution for the entire expression; if none of the solutions are clearly better than the others, an ambiguity error is diagnosed. Otherwise, we proceed as if the solver only found the best solution. The final step applies the solution to the expression by replacing type variables appearing in the types of sub-expressions with their fixed types.

The utmost care must be taken when working with type variables because unlike all other types, they are not allocated with indefinite lifetime. Type variables live in the constraint solver arena, which grows and shrinks as the solver explores branches of the solution space. Types that *contain* type variables, and other structures that recursively contain such types, also need to be allocated in the constraint solver arena. Type variables "escaping" from the constraint solver can crash the compiler in odd ways. Assertions should be used to rule out type variables from appearing in the wrong places.

The printed representation of a type variable is `$Tn`, where `n` is an incrementing integer local to the constraint system. One way you can see type variables in action is by passing the `-Xfrontend -debug-constraints` compiler flag.

L-value types An l-value type represents the type of an expression appearing on the left hand side of an assignment operator (hence the "l" in l-value), or as an argument to an `inout` parameter in a function call. L-value types wrap an *object type* which is the type of the stored value; they print out as `@lvalue T` where `T` is the object type, but this is not valid syntax in the language.

L-value types appear in type-checked assignment expressions and call arguments for `inout` parameters. If you're familiar with C++, you can think of an l-value type as somewhat analogous to a C++ mutable reference type "`T &`"—unlike C++ though, they are not directly visible in the source language.

Error types Error types are returned when type substitution encounters an invalid or missing conformance (Chapter 7). In this case, the error type wraps the original type, and prints as the original type to make types coming from malformed conformances more readable in diagnostics.

The expression type checker also assigns error types to invalid declaration references. This uses the singleton form of the error type, which prints as `<<error type>>`. To avoid user confusion, diagnostics containing the singleton error type should not be emitted. Generally, any expression whose type contains an error type does not need to be diagnosed, because a diagnostic should have been emitted elsewhere.

3.6. Source Code Reference

Key source files:

- `include/swift/AST/Type.h`
- `include/swift/AST/Types.h`
- `lib/AST/Type.cpp`

Other source files:

- `include/swift/AST/TypeNodes.def`
- `include/swift/AST/TypeVisitor.h`
- `include/swift/AST/CanTypeVisitor.h`

`Type` *class*

Represents an immutable, uniqued type. Meant to be passed as a value, it stores a single instance variable, a `TypeBase *` pointer.

The `getPointer()` method returns this pointer. The pointer is not `const`, however neither `TypeBase` nor any of its subclasses define any mutating methods. The pointer may be `nullptr`; the default constructor `Type()` constructs an instance of a null type. Most methods will crash if called on a null type; only the implicit `bool` conversion and `getPointer()` are safe.

The `getPointer()` method is only used occasionally, because types are usually passed as `Type` and not `TypeBase *`, and `Type` overloads `operator->` to forward method calls to the `TypeBase *` pointer. While most operations on types are actually methods on `TypeBase`, a few methods are also defined on `Type` itself (these are called with "`.`" instead of "`->`").

Various traversals `walk()` is a general pre-order traversal where the callback returns a tri-state value—continue, stop, or skip a sub-tree. Built on top of this are two simpler variants; `findIf()` takes a boolean predicate, and `visit()` takes a void-returning callback which offers no way to terminate the traversal.

Transformations `transformWithPosition()`, `transformRec()`, `transform()`. As with the traversals, the first of the three is the most general, and the other two are built on top. In all three cases, the callback is invoked on all types contained within a type, recursively. It can either elect to replace a type with a new type, or leave a type unchanged and instead try to transform any of its child types.

Substitution `subst()` implements type substitution, which is a particularly common kind of transform which replaces generic parameters or archetypes with concrete types (Section 7.5).

Printing `print()` outputs the string form of a type, with many customization options; `dump()` prints the tree structure of a type in an s-expression form. The latter is extremely useful for invoking from inside a debugger, or ad-hoc print debug statements.

The `Type` class explicitly deletes the overloads of `operator==` and `operator!=` to make the choice between pointer and canonical equality explicit. To check pointer equality of possibly-sugared types, first unwrap both sides with a `getPointer()` call:

```
1 if (lhsType.getPointer() == rhsType.getPointer())
2   ...;
```

The more common canonical type equality check is implemented by the `isEqual()` method on `TypeBase`:

```
1 if (lhsType->isEqual(rhsType))
2   ...;
```

`TypeBase` *class*

The root of the type kind hierarchy. Its instances are always uniqued and allocated by the AST context, either the permanent arena or constraint solver arena. Instances are usually wrapped in `Type`. The various subclasses correspond to the different kinds of types:

- `NominalType` and its four subclasses:
  - `StructType`,
  - `EnumType`,
  - `ClassType`,
  - `ProtocolType`.
- `BoundGenericNominalType` and its three subclasses:
  - `BoundGenericStructType`,
  - `BoundGenericEnumType`,
  - `BoundGenericClassType`.
- The structural types `TupleType`, `MetatypeType`.
- `AnyFunctionType` and its two subclasses:
  - `FunctionType`,
  - `GenericFunctionType`.
- `GenericTypeParamType`, `DependentMemberType`, the two type parameter types.
- `ArchetypeType`, and its three subclasses:
  - `PrimaryArchetypeType`,
  - `OpenedArchetypeType`,
  - `OpaqueArchetypeType`.
- The abstract types:
  - `ProtocolCompositionType`,

  - `ParameterizedProtocolType`,
  - `ExistentialType`,
  - `ExistentialMetatypeType`,
  - `DynamicSelfType`.

- `SugarType` and its four subclasses:
  - `TypeAliasType`,
  - `OptionalType`,
  - `ArrayType`,
  - `DictionaryType`.

- `BuiltinType` and its subclasses (there are a bunch of esoteric ones; only a few are shown below):
  - `BuiltinRawPointerType`,
  - `BuiltinVectorType`,
  - `BuiltinIntegerType`,
  - `BuiltinIntegerLiteralType`,
  - `BuiltinNativeObjectType`,
  - `BuiltinBridgeObjectType`.

- `ReferenceStorageType` and its two subclasses:
  - `WeakStorageType`,
  - `UnownedStorageType`.

- Miscellaneous types:
  - `UnboundGenericType`,
  - `PlaceholderType`,
  - `TypeVariableType`,
  - `LValueType`,
  - `ErrorType`.

Each concrete subclass defines some set of static factory methods, usually named `get()` or similar, which take the structural components and construct a new, uniqued type of this kind. There are also getter methods, prefixed with `get`, which project the structural components of each kind of type. It would be needlessly duplicative to list all of the getter methods for each subclass of `TypeBase`; you can pursue them yourself by looking at `include/swift/AST/Types.h`.

Dynamic casts Subclasses of `TypeBase *` are identifiable at runtime via the `is<>`, `castTo<>` and `getAs<>` template methods. To check if a type has a specific kind, use `is<>`:

```
Type type = ...;

if (type->is<FunctionType>())
  ...;
```

To conditionally cast a type to a specific kind, use `getAs<>`, which returns `nullptr` if the cast fails:

```
if (FunctionType *funcTy = type->getAs<FunctionType>())
  ...;
```

Finally, `castTo<>` is an unconditional cast which asserts that the type has the required kind:

```
FunctionType *funcTy = type->castTo<FunctionType>();
```

These template methods desugar the type if it is a sugared type, and the casted type can never itself be a sugared type. This is usually what you want; for example if `type` is the `Swift.Void` type alias type, then `type->is<TupleType>()` returns true, because it is for all intents and purposes a tuple (an empty tuple), except when printed in diagnostics.

There are also top-level template functions `isa<>`, `dyn_cast<>` and `cast<>` that operate on `TypeBase *`. Using these with `Type` is an error; you must explicitly unwrap the pointer with `getPointer()`. These casts do not desugar, and permit casting to sugared types. This is occasionally useful if you need to handle sugared types differently from canonical types for some reason:

```
Type type = ...;

if (isa<OptionalType>(type.getPointer()))
  ...;
```

Canonical types The `getCanonicalType()` method outputs a `CanType` wrapping the canonical form of this `TypeBase *`. The `isCanonical()` method checks if a type is canonical.

Visitors If you need to exhaustively handle each kind of type, the simplest way is to switch over the kind, which is an instance of the `TypeKind` enum, like this:

```
Type ty = ...;
switch (ty->getKind()) {
case TypeKind::Struct: {
  auto *structTy = ty->castTo<StructType>();
  ...
}
case TypeKind::Enum:
  ...
case TypeKind::Class:
  ...
}
```

However, in most cases it is more convenient to use the *visitor pattern* instead. You can subclass `TypeVisitor` and override various `visit`*`Kind`*`Type()` methods, then hand the type to the visitor's `visit()` method, which performs the switch and dynamic cast dance above:

```
class MyVisitor: public TypeVisitor<MyVisitor> {
public:
  void visitStructType(StructType *ty) {
    ...
  }
};

MyVisitor visitor;

Type ty = ...;
visitor.visit(ty);
```

The `TypeVisitor` also defines various methods corresponding to abstract base classes in the `TypeBase` hierarchy, so for example you can override `visitNominalType()` to handle all nominal types at once.

The `TypeVisitor` preserves information if it receives a sugared type; for example, visiting `Int?` will call `visitOptionalType()`, while visiting `Optional<Int>` will call `visitBoundGenericEnumType()`. In the common situation where the semantics of your operation do not depend on type sugar, you can use the `CanTypeVisitor` template class instead. Here, the `visit()` method takes a `CanType`, so `Int?` will need to be canonicalized to `Optional<Int>` before being passed in.

Nominal types A handful of methods on `TypeBase` exist which perform a desugaring cast to a nominal type (so they will also accept a type alias type or other sugared type), and return the nominal type declaration, or `nullptr` if the type isn't of a nominal kind:

- `getAnyNominal()` returns the nominal type declaration of `UnboundGenericType`, `NominalType` or `BoundGenericNominalType`.
- `getNominalOrBoundGenericNominal()` returns the nominal type declaration of a `NominalType` or `BoundGenericNominalType`.
- `getStructOrBoundGenericStruct()` returns the type declaration of a `StructType` or `BoundGenericStructType`.
- `getEnumOrBoundGenericEnum()` returns the type declaration of an `EnumType` or `BoundGenericEnumType`.
- `getClassOrBoundGenericClass()` returns the class declaration of a `ClassType` or `BoundGenericClassType`.
- `getNominalParent()` returns the parent type stored by an `UnboundGenericType`, `NominalType` or `BoundGenericNominalType`.

Recursive properties Various predicates are computed when a type is constructed and are therefore cheap to check:

- `hasTypeVariable()` determines whether the type was allocated in the permanent arena or the constraint solver arena.
- `hasArchetype()`, `hasOpaqueArchetype()`, `hasOpenedExistential()`.
- `hasTypeParameter()`.
- `hasUnboundGenericType()`, `hasDynamicSelf()`, `hasPlaceholder()`.
- `isLValue()`—despite the "`is`" in the name, this is a recursive property and not the same as `ty->is<LValueType>()`.

Utility operations These encapsulate frequently-useful patterns.

- `getOptionalObjectType()` returns the type `T` if the type is some `Optional<T>`, otherwise it returns the null type.
- `getMetatypeInstanceType()` returns the type `T` if the type is some `T.Type`, otherwise it returns `T`.
- `mayHaveMembers()` answers if this is a nominal type, archetype, existential type or dynamic Self type.

Recovering the AST context All non-canonical types point at their canonical type, and canonical types point at the AST context.

- `getASTContext()` returns the singleton AST context from a type.

`CanType` *class*

The `CanType` class wraps a `TypeBase *` pointer which is known to be canonical. The pointer can be recovered with the `getPointer()` method. It forwards various methods to either `Type` or `TypeBase *`. There is an implicit conversion from `CanType` to `Type`. In the other direction, the explicit one-argument constructor `CanType(Type)` asserts that the type is canonical; however, most of the time the `getCanonicalType()` method on `TypeBase` is used instead.

The `operator==` and `operator!=` operators are used to test `CanType` for pointer equality. The `isEqual()` method described earlier implements canonical equality on sugared types by first canonicalizing both sides, and then checking the resulting canonical types for pointer equality. Therefore, the following are equivalent:

```
1 if (lhsType->isEqual(rhsType)) ...;
2 if (lhsType->getCanonicalType() == rhsType->getCanonicalType()) ...;
```

The `CanType` class can be used with the `isa<>`, `cast<>` and `dyn_cast<>` templates. Instead of returning the actual `TypeBase` subclass, the latter two return a *canonical type wrapper* for that subclass. Every subclass of `TypeBase` has a corresponding canonical type wrapper; if the subclass is named `FooType`, the canonical wrapper is named `CanFooType`. Canonical type wrappers forward `operator->` to the specific `TypeBase` subclass, and define methods of their own (called with ".") which project the known-canonical components of the type.

For example, `FunctionType` has a `getResult()` method returning `Type`, so the canonical type wrapper `CanFunctionType` has a `getResult()` method returning a `CanType`. The wrapper methods are not exhaustive, and their use is not required because you can instead make explicit calls to `CanType(Type)` or `getCanonicalType()` after projecting a type that is known to be canonical.

```
1 CanType canTy = ...;
2 CanFunctionType canFuncTy = cast<FunctionType>(canTy);
3
4 // method on CanFunctionType: returns CanType(canFuncTy->getResult())
5 CanType canResultTy = canFuncTy.getResult();
6
7 // operator-> forwards to method on FunctionType: returns Type
8 CanType resultTy = CanType(canFuncTy->getResult());
```

`AnyFunctionType` *class*

This is the base class of `FunctionType` and `GenericFunctionType`.

- `getParams()` returns an array of `AnyFunctionType::Param`.
- `getResult()` returns the result type.
- `getExtInfo()` returns an instance of `AnyFunctionType::ExtInfo` storing the additional non-type attributes.

`AnyFunctionType::Param` *class*

This represents a parameter in a function type's parameter list.

- `getPlainType()` returns the type of the parameter. If the parameter is variadic (`T...`), this is the element type `T`.
- `getParameterType()` same as above, but if the parameter is variadic, returns the type `Array<T>`.
- `isVariadic()`, `isAutoClosure()` are the special behaviors.
- `getValueOwnership()` returns an instance of the `ValueOwnership` enum.

`ValueOwnership` *enum class*

The possible ownership attributes on a function parameter.

- `ValueOwnership::Default`
- `ValueOwnership::InOut`
- `ValueOwnership::Shared`
- `ValueOwnership::Owned`

`AnyFunctionType::ExtInfo` *class*

This represents the non-type attributes of a function type.

4. Declarations

The different kinds of declarations are categorized into a taxonomy. A *value declaration* has a name that can be directly referenced from an expression. Each value declaration also has an *interface type*. Roughly speaking, this is the type of an expression referencing the declaration. Most declarations are value declarations, but there are some important exceptions. Extensions, described in Chapter 12, add members to a type but do not themselves have names. A *top-level code declaration* is another kind of declaration that is not a value declaration; it holds the statements and expressions at the top level of a source file.

A *type declaration* is an important kind of value declaration. A type declaration declares a new type that you can write down in a type annotation; this is the *declared interface type* of the type declaration. Since type declarations are value declarations, they also have an interface type, which is the type of an expression referencing the type declaration. When a type is used as a value, the type of the value is a metatype. A type declaration's interface type is therefore the metatype of its declared interface type.

Struct, enum and class declarations are called *nominal type declarations*. Protocols are also nominal type declarations, but they are special enough it is best to think of them as a separate kind of entity.

The *self interface type* of a type or extension declaration is the type from which the `self` parameter type of a method is derived. In a struct, enum or class declaration, the self interface type and declared interface type coincide. In a protocol, the self interface type is the protocol `Self` type (Section 5.4).

In the following, the nominal type declaration `Fish` is referenced twice, first as a type annotation, and then in an expression:

```
1 struct Fish {}
2
3 let myFish: Fish = Fish()
```

This is a very simple piece of code, but there's more going on than seems at first glance. The first occurrence of `Fish` is the type annotation for the variable declaration `myFish`, so the interface type of `myFish` becomes the nominal type `Fish`. The second occurrence is inside the initial value expression of `myFish`. The callee of the call expression `Fish()` is the type expression `Fish`, whose type is the metatype `Fish.Type`. A call of a metatype is transformed into a call of the `init` member, which names a constructor declaration.

Table 4.1.: Classifying various entities in our taxonomy

Entity kind	Decl?	Value decl?	Type decl?	Decl context?
Module	✓	✓	✓	✓
Source file	×	×	×	✓
Nominal type	✓	✓	✓	✓
Extension	✓	×	×	✓
Generic parameter	✓	✓	✓	×
Function	✓	✓	×	✓
Variable	✓	✓	×	×
Top-level code	✓	×	×	✓
Closure expression	×	×	×	✓
Call expression	×	×	×	×

Constructors are called on an instance of the metatype of a type, and return an instance of the type. So the initial value expression has the type `Fish`, which matches the interface type of `myFish`. The constructor has the interface type `(Fish.Type) -> () -> Fish`.

A *declaration context* is an entity that can contain declarations. Declaration contexts are distinct from declarations. Module declarations, nominal type declarations, extension declarations and function declarations are also declaration contexts. Not all declarations are declaration contexts; variable declarations and generic parameter declarations are not. Furthermore, some declaration contexts are not declarations. A closure expression is not a declaration, but it is a declaration context, because the body of a closure can contain variable, function and type declarations. A source file is another kind of declaration context that is not a declaration. A summary of the examples so far is shown in Table 4.1.

Declarations and declaration contexts are nested within each other. The roots in this hierarchy are module declarations; all other declarations and declaration contexts point at a parent declaration context. Source files are always immediate children of module declarations.

A *local context* is any declaration context that is not a module, source file, type declaration or extension. Top-level code declarations, function declarations and closure expressions are three kinds of local contexts we've already seen.

The three remaining kinds of local context are subscript declarations, enum element declarations and initializer contexts:

- Subscripts and enum elements are local contexts, because they contain their parameter declarations.
- Subscript declarations can also be generic, so they need to contain their generic parameters.

- Initializer contexts represent the initial value expression of a variable that is itself not a child of a local context. This ensures that any declarations appearing in the initial value expression of a variable are always children of a local context.

There is special terminology for type declarations in different kinds of declaration contexts:

- A *top-level type* is an immediate child of a source file.
- A *nested type* or *member type* is an immediate child of a nominal type declaration or an extension.
- A *local type* is an immediate child of a local context.

Similarly, for functions:

- A *top-level function* or *global function* is an immediate child of a source file.
- A *method* is an immediate child of a nominal type declaration or an extension.
- A *local function* is an immediate child of a local context.

And finally, for variables:

- A *global variable* is an immediate child of a source file.
- A *property* is an immediate child of a nominal type declaration or an extension.
- A *local variable* is an immediate child of a local context.

4.1. Type Declarations

Struct, enum and class declarations These are the concrete nominal types. The declared interface type of a non-generic nominal type declaration is a nominal type. If the nominal type declaration is generic, the declared interface type is a generic nominal type where the generic arguments are the declaration's generic parameters.

Concrete nominal types can be nested inside of other declaration contexts, with a few limitations described in Section 7.4. The declared interface type reflects this nesting. For example, the declared interface type of `Outer.Inner` is the generic nominal type `Outer<T>.Inner<U>`:

```
1 struct Outer<T> {
2   struct Inner<U> {}
3 }
```

Classes can inherit from other classes; Chapter 16 describes how inheritance interacts with generics.

Protocol declarations The declared interface type of a protocol declaration is the protocol type `P`. Protocols are the fourth kind of nominal type, but they behave differently in many ways, because they do not have concrete instances. Protocol declarations are described in Chapter 5.4.

Type alias declarations Type aliases assign a new name to an underlying type. The declared interface type is a type alias type whose canonical type is the underlying type of the type alias. The special case of type aliases in protocols is discussed in Section 10.3.

Generic parameter declarations Generic parameter declarations appear inside generic parameter lists of generic declarations. The declared interface type of a generic parameter declaration is the sugared generic parameter type that prints as the name of the declaration. The canonical type of this type is the generic parameter type `τ_d_i`, where `d` is the depth and `i` is the index. Generic parameter declarations are described in Chapter 5.

Associated type declarations Associated type declarations appear inside protocols. The declared interface type of an associated type `A` is a bound dependent member type `Self.[P]A` referencing the declaration of `A`, with the `Self` generic parameter of the protocol as the base type. Associated type declarations are described in Section 5.4.

4.2. Function Declarations

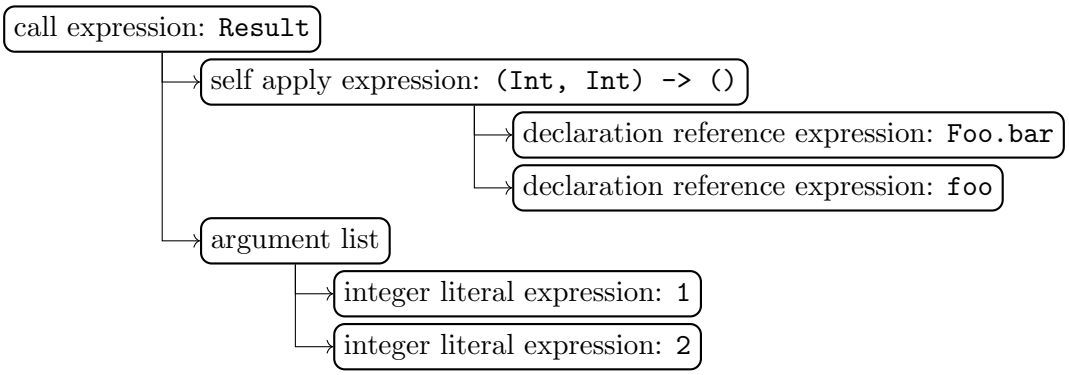
Function declarations Functions can either appear at the top level, inside of a local context such as another function, or as a member of a type, called a method. If a function is itself generic or nested inside of a generic context, the interface type is a generic function type, otherwise it is a function type.

The interface type of of a function is constructed from the interface types of the function's parameter declarations, and the function's return type. If the return type is omitted, it becomes the empty tuple type `()`. For methods, this function type is then wrapped in another level of function type representing the base of the call which becomes the `self` parameter of the method.

The `self` parameter's type and parameter flags are constructed from the self interface type of the method's type declaration, and various attributes of the method:

- If the method is `mutating`, the `self` parameter becomes `inout`.
- If the method returns the dynamic Self type, the `self` parameter type is wrapped in the dynamic Self type.
- Finally, if the method is `static`, the `self` parameter is wrapped in a metatype.

Figure 4.1.: Two levels of function application in a method call `foo.bar(1, 2)`



This can be summarized as follows; note that `(Self)` parameter list means the self interface type of the method's type declaration, together with any additional parameter flags computed via the above:

Generic?	Method?	Interface type
×	×	`(Params...) -> Result`
✓	×	`<Sig...> (Params...) -> Result`
×	✓	`(Self) -> (Params...) -> Result`
✓	✓	`<Sig...> (Self) -> (Params...) -> Result`

The two levels of function type in the interface type of a method mirror the two-level structure of a method call expression `foo.bar(1, 2)`, shown in Figure 4.1:

- The self apply expression `foo.bar` applies the single argument `foo` to the method's `self` parameter. The type of the self apply expression is the method's inner function type.
- The outer call applies the argument list `(1, 2)` to the inner function type. The type of the outer call expression is the method's return type.

Constructor declarations Constructor declarations always appear as members of other types, and are named `init`. The interface type of a constructor takes a metatype and returns an instance of the constructed type, possibly wrapped in an `Optional`.

Generic?	Interface type
×	`(Self.Type) -> (Params...) -> Self`
✓	`<Sig...> (Self.Type) -> (Params...) -> Self`

Class constructors also have a *initializer interface type*, used when a subclass initializer delegates to an initializer in the superclass. The initializer interface type is the same as the interface type, except it takes a self value instead of a self metatype.

Generic?	Initializer interface type
×	`(Self) -> (Params...) -> Self`
✓	`<Sig...> (Self) -> (Params...) -> Self`

Destructor declarations Destructor declarations cannot have a generic parameter list, a `where` clause, or a parameter list. Formally they take no parameters and return `()`.

Generic?	Interface type
×	`(Self) -> () -> ()`
✓	`<Sig...> (Self) -> () -> ()`

4.3. Storage Declarations

Storage declarations represent the declaration of an l-value. Storage declarations can have zero or more associated accessor declarations. The accessor declarations are siblings of the storage declaration in the declaration context hierarchy.

Variable declarations The interface type of a variable is the stored value type, possibly wrapped in a reference storage type if the variable is declared as `weak` or `unowned`. The *value interface type* of a variable is the storage type without any wrapping.

For historical reasons, the interface type of a property (a variable appearing inside of a type) does not include the `Self` clause, the way that method declarations do.

Variable declarations are always created alongside a *pattern binding declaration* which represents the various ways in which variables can be bound to values in Swift. A pattern binding declaration consists of one or more *pattern binding entries*. Each pattern binding entry has a *pattern* and an optional *initial value expression*. A pattern declares zero or more variables.

Example 4.1. A pattern binding declaration with a single entry, where the pattern declares a single variable:

```
1 let x = 123
```

Same as the above, except with a more complex pattern which declares a variable storing the first element of a tuple while discarding the second element:

```
1 let (x, _) = (123, "hello")
```

A pattern binding declaration with a single entry, where the pattern declares two variables `x` and `y`:

```
let (x, y) = (123, "hello")
```

A pattern binding declaration with two entries, where the first pattern declares `x` and the second declares `y`:

```
let x = 123, y = "hello"
```

A pattern binding declaration with a single entry that does not declare any variables:

```
let _ = ignored()
```

And finally, two pattern binding declarations, where each one pattern binding declaration has a single entry declaring a single variable:

```
let x = 123
let y = "hello"
```

Example 4.2. If the pattern binding declaration appears outside of a local context, each entry must declare at least one variable, so both pattern binding declarations are rejected here:

```
let _ = 123

struct S {
  let _ = "hello"
}
```

Example 4.3. A funny quirk of the pattern grammar is that typed patterns and tuple patterns do not compose in the way one might think. If "`let x: Int`" is a typed pattern declaring a variable `x` type with annotation `Int`, and "`let (x, y)`" is a tuple pattern declaring two variables `x` and `y`, you might expect "`let (x: Int, y: Int)`" to declare two variables `x` and `y` with type annotations `Int` and `String` respectively; what actually happens is you get a tuple pattern declaring two variables named `Int` and `String` that binds a two-element tuple with *labels* `x` and `y`:

```
let (x: Int, y: String) = (x: 123, y: "hello")
print(Int)  // huh? prints 123
print(String)  // weird! prints "hello"
```

Parameter declarations Functions, enum elements and subscripts can have parameter lists; each parameter is represented by a parameter declaration. The interface type of a declaration with a parameter list is built by first computing the interface type of each parameter. Closure expressions also have parameter lists and thus parent parameter declarations.

Among other things, the parameter declaration stores the value ownership kind, the variadic flag, and the `@autoclosure` attribute. This is in fact the same exact information encoded in the parameter list of a function type.

Parameter declarations of named declarations can also have argument labels and default argument expressions, which are not encoded in a function type. These phenomena are only visible when directly calling a named declaration and not a closure value.

Subscript declarations Subscripts always appear as members of types, with a special declaration name. The interface type of a subscript is a function type taking the index parameters and returning the storage type. The value interface type of a subscript is just the storage type.

For historical reasons, the interface type of a subscript does not include the `Self` clause, the way that method declarations do.

Generic?	Interface type
×	`(Indices...) -> Value`
✓	`<Sig...> (Indices...) -> Value`

Accessor declarations The interface type of an accessor depends the accessor kind. For example, getters return the value, and setters take the new value as a parameter. Property accessors do not take any other parameters; subscript accessors also take the subscript's index parameters. There is a lot more to say about accessors and storage declarations, but unfortunately, you'll have to wait for the next book.

4.4. Source Code Reference

Key source files:

- `include/swift/AST/Decl.h`
- `include/swift/AST/DeclContext.h`
- `lib/AST/Decl.cpp`
- `lib/AST/DeclContext.cpp`

Other source files:

- include/swift/AST/DeclNodes.def
- include/swift/AST/ASTVisitor.h
- include/swift/AST/ASTWalker.h

`Decl` *class*

Base class of declarations. Figure 4.2 shows various subclasses, which correspond to the different kinds of declarations defined previously in this chapter.

Instances are always allocated in the permanent arena of the `ASTContext`, either when the declaration is parsed or synthesized. The top-level `isa<>`, `cast<>` and `dyn_cast<>` template functions support dynamic casting from `Decl *` to any of its subclasses.

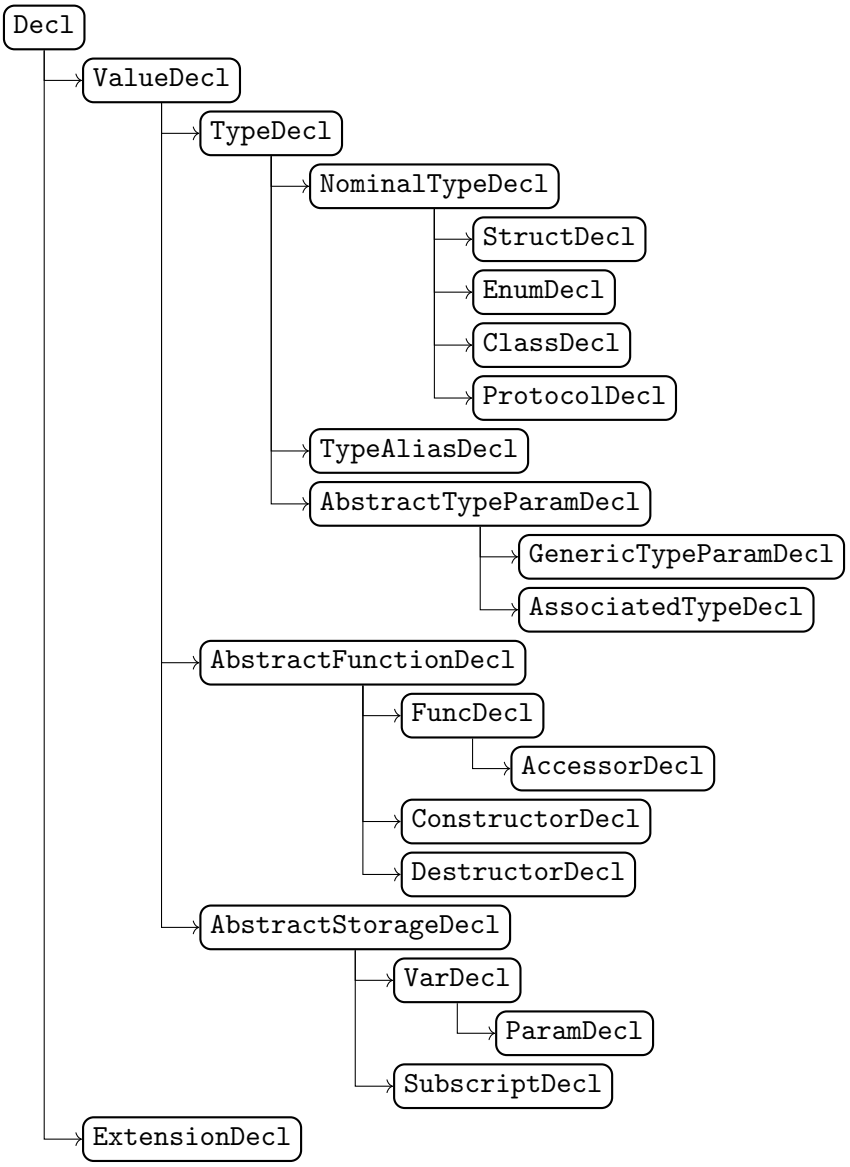
- `getDeclContext()` returns the parent `DeclContext` of this declaration.
- `getInnermostDeclContext()` if this declaration is also a declaration context, returns the declaration as a `DeclContext`, otherwise returns the parent `DeclContext`.
- `getASTContext()` returns the singleton AST context from a declaration.

Visitors If you need to exhaustively handle each kind of declaration, the simplest way is to switch over the kind, which is an instance of the `DeclKind` enum, like this:

```
Decl *decl = ...;
switch (decl->getKind()) {
case DeclKind::Struct: {
  auto *structDecl = decl->castTo<StructDecl>();
  ...
}
case DeclKind::Enum:
  ...
case DeclKind::Class:
  ...
}
```

However, just as with types, is can be more convenient to use the visitor pattern. You can subclass `ASTVisitor` and override various `visit`*`Kind`*`Decl()` methods, then hand the declaration to the visitor's `visit()` method, which performs the switch and dynamic cast dance above:

Figure 4.2.: The `Decl` class hierarchy



```
class MyVisitor: public ASTVisitor<MyVisitor> {
public:
  void visitStructDecl(StructType *decl) {
    ...
  }
};

MyVisitor visitor;

Decl *decl = ...;
visitor.visit(decl);
```

The `ASTVisitor` also defines various methods corresponding to abstract base classes in the `Decl` hierarchy, so for example you can override `visitNominalTypeDecl()` to handle all nominal type declarations at once. The `ASTVisitor` is more general than just visiting declarations; it also supports visiting statements, expressions, and type representations.

A more elaborate form is implemented by the `ASTWalker`. While the visitor visits a single declaration, the walker traverses nested declarations, statements and expressions for you in a pre-order walk.

`ValueDecl` *class*

Base class of named declarations.

- `getDeclName()` returns the declaration's name.
- `getInterfaceType()` returns the declaration's interface type.

`TypeDecl` *class*

Base class of type declarations.

- `getDeclaredInterfaceType()` returns the type of an instance of this declaration.

`NominalTypeDecl` *class*

Base class of nominal type declarations. Also a `DeclContext`.

- `getSelfInterfaceType()` returns the type of the `self` value inside the body of this declaration. Different from the declared interface type for protocols, where the declared interface type is a nominal but the declared self type is the generic parameter `Self`.

- `getDeclaredType()` returns the type of an instance of this declaration, without generic arguments. If the declaration is generic, this is an unbound generic type. If this declaration is not generic, this is a nominal type. This is occasionally used in diagnostics instead of the declared interface type, when the generic parameter types are irrelevant.

`TypeAliasDecl` *class*

A type alias declaration. Also a `DeclContext`.

- `getDeclaredInterfaceType()` returns the underlying type of the type alias declaration, wrapped in type alias type sugar.
- `getUnderlyingType()` returns the underlying type of the type alias declaration, without wrapping it in type alias type sugar.

`AbstractFunctionDecl` *class*

Base class of function-like declarations. Also a `DeclContext`.

- `getImplicitSelfDecl()` returns the implicit `self` parameter, if there is one.
- `getParameters()` returns the function's parameter list.
- `getMethodInterfaceType()` returns the type of a method without the `Self` clause.
- `getResultInterfaceType()` returns the return type of this function or method.

`ParameterList` *class*

The parameter list of `AbstractFunctionDecl`, `EnumElementDecl` or `SubscriptDecl`.

- `size()` returns the number of parameters.
- `get()` returns the `ParamDecl` at the given index.

`ConstructorDecl` *class*

Constructor declarations.

- `getInitializerInterfaceType()` returns the initializer interface type, used when type checking `super.init()` delegation.

`AbstractStorageDecl` *class*

Base class for storage declarations.

- `getValueInterfaceType()` returns the type of the stored value, without `weak` or `unowned` storage qualifiers.

`DeclContext` *class*

Base class for declaration contexts. The top-level `isa<>`, `cast<>` and `dyn_cast<>` template functions also support dynamic casting from a `DeclContext *` to any of its subclasses.

There are a handful of subclasses which are not also subclasses of `Decl *`:

- `ClosureExpr`.
- `FileUnit` and its various subclasses, such as `SourceFile`.
- A few other less interesting ones you can find in the source.

Utilities for understanding the nesting of declaration contexts:

- `getAsDecl()` if declaration context is also a declaration, returns the declaration, otherwise returns `nullptr`.
- `getParent()` returns the parent declaration context.
- `isModuleScopeContext()` returns true if this is a `ModuleDecl` or `FileUnit`.
- `isTypeContext()` returns true if this is a nominal type declaration or an extension.
- `isLocalContext()` returns true if this is not a module scope context or type context.
- `getParentModule()` returns the module declaration at the root of the hierarchy.
- `getModuleScopeContext()` returns the innermost parent which is a `ModuleDecl` or `FileUnit`.
- `getParentSourceFile()` returns the innermost parent which is a source file, or `nullptr` if this declaration context was not parsed from source.
- `getInnermostDeclarationDeclContext()` returns the innermost parent which is also a declaration, or `nullptr`.
- `getInnermostDeclarationTypeContext()` returns the innermost parent which is also a nominal type or extension, or `nullptr`.

Operations on type contexts:

- `getSelfNominalDecl()` returns the nominal type declaration if this is a type context, or `nullptr`.

- `getSelfStructDecl()` as above but result is a `StructDecl *` or `nullptr`.
- `getSelfEnumDecl()` as above but result is a `EnumDecl *` or `nullptr`.
- `getSelfClassDecl()` as above but result is a `ClassDecl *` or `nullptr`.
- `getSelfProtocolDecl()` as above but result is a `ProtocolDecl *` or `nullptr`.
- `getDeclaredInterfaceType()` delegates to the method on `NominalTypeDecl` or `ExtensionDecl` as appropriate.
- `getSelfInterfaceType()` is similar.

Generics-related methods on `DeclContext` are described in Section 5.5.

5. Generic Declarations

A *generic declaration* is a declaration with a generic parameter list. The following kinds of declarations can be generic:

- classes, structs and enums,
- type aliases,
- functions,
- constructors,
- subscripts.

Generic type aliases were introduced in Swift 3 [18]. Generic subscripts were introduced in Swift 4 [19].

The *parsed* generic parameter list of a declaration is the subset of generic parameter declarations written in source only, with the `<...>` syntax following the declaration name. The declaration's generic parameter list includes the parsed generic parameter list together with any implicit generic parameters:

1. Functions and subscripts may have a parsed generic parameter list, or they can declare opaque parameters with the `some` keyword, or both (Section 5.3).
2. Protocols always have a single implicit `Self` generic parameter, and no parsed generic parameter list (Section 5.4).
3. Extensions always have an implicit set of generic parameters inherited from the extended type, and no parsed generic parameter list (Chapter 12).

Parsed generic parameters, the protocol `Self` type, and the implicit generic parameters of an extension all have names that remain in scope for the entire source range of the generic declaration. Generic parameters introduced by opaque parameter declarations are unnamed; only the value declared by the opaque parameter has a name.

All generic declarations are declaration contexts, because they contain their generic parameter declarations. A *generic context* is a declaration context where at least one parent context is a generic declaration. Note the subtle distinction in the meaning of "generic" when talking about declarations and declaration contexts; a declaration is

generic only if it has generic parameters of its own, whereas a declaration context being a generic context is a transitive properly inherited from the parent context.

Inside a generic context, unqualified name lookup will find all outer generic parameters. Each generic parameter is therefore uniquely identified within a generic context by its *depth* and *index*:

- The depth identifies a specific generic declaration, starting from zero for the top-level generic declaration and incrementing for each nested generic declaration.
- The index identifies a generic parameter within a single generic parameter list.

The declared interface type of a generic parameter declaration is a sugared type that prints as the generic parameter name. The canonical type of this type only stores the depth and index. The notation for a canonical generic parameter type is `τ_d_i`, where `d` is the depth and `i` is the index.

Example 5.1. Listing 5.1 declares a `LinkedList` type with a single generic parameter named `Element`, and a `mapReduce()` method with two generic parameters named `T` and `A`. All three generic parameters are visible from inside the method:

Name	Depth	Index	Canonical type
`Element`	0	0	`τ_0_0`
`T`	1	0	`τ_1_0`
`A`	1	1	`τ_1_1`

5.1. Constraint Types

A generic requirement adds new capabilities to a generic parameter type, by restricting the possible substituted concrete types to those that provide this capability. The next section will introduce the trailing `where` clause syntax for stating generic requirements in a fully general way. Before doing that, we'll take a look at the simpler mechanism of stating a *constraint type* in the inheritance clause of a generic parameter declaration:

```
1 func allEqual<T: Equatable>(_ elements: [T]) {...}
```

A constraint type is one of the following:

1. A protocol type, like `Hashable`.
2. A parameterized protocol type, like `Sequence<String>` (Section 5.4).
3. A protocol composition, like `ShapeProtocol & MyClass`. Protocol compositions were originally just compositions of protocol types, but they can include class types as of Swift 4 [20].

Listing 5.1.: Two nested generic declarations

```
enum LinkedList<Element> {
  case none
  indirect case entry(Element, LinkedList<Element>)

  func mapReduce<T, A>(_ f: (Element) -> T,
                       _ m: (A, T) -> A,
                       _ a: A) -> A {
    switch self {
    case .none:
      return a
    case .entry(let x, let xs):
      return m(xs.mapReduce(f, m, a), f(x))
  }
}
```

4. A class type, like `NSObject`.

5. The `AnyObject` *layout constraint*, which restricts the possible concrete types to those represented as a single reference-counted pointer.

6. The empty protocol composition, written `Any`. Writing `Any` in a generic parameter's inheritance clause is pointless, but it is allowed for completeness.

Constraint types can appear in various positions:

1. In the inheritance clause of a generic parameter declaration, which is the focus of this section.

2. On the right hand side of a conformance, superclass or layout requirement in a `where` clause, which you will see shortly.

3. In the inheritance clauses of protocols and associated types (Section 5.4).

4. Following the `some` keyword in an opaque parameter (Section 5.3) or return type (Chapter 14).

5. Following the `any` keyword in an existential type (Chapter 15). A single class type cannot be the constraint type of an existential; `any NSObject` is just written as `NSObject`. Existential types where the constraint type is `AnyObject` and `Any` can also be written without the `any` keyword.

Listing 5.2.: The constraint type of `B` in `open(box:)` refers to `C`

```
class Box<Contents> {
  var contents: Contents
}

func open<B: Box<C>, C>(box: B) -> C {
  return box.contents
}

struct Vegetables {}
class FarmBox: Box<Vegetables> {}

let vegetables: Vegetables = open(box: FarmBox())
```

Example 5.2. Listing 5.2 exhibits a generic parameter whose constraint type references another generic parameter visible from the current scope. The generic parameter `C` is visible in the entire declaration of `open(box:)`, including the generic parameter list.

5.2. Requirements

A constraint type in the inheritance clause of a generic parameter declaration is syntax sugar for a `where` clause with a single entry whose subject type is the generic parameter type:

```
struct Set<Element: Hashable> {...}
struct Set<Element> where Element: Hashable {...}
```

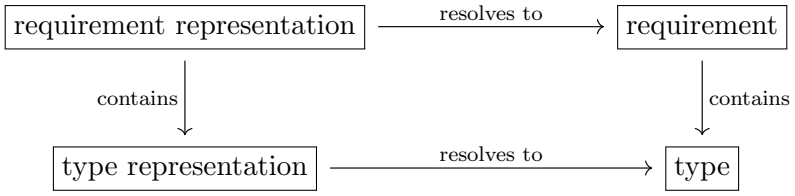
The requirements in a `where` clause state the subject type explicitly, allowing stating requirements on dependent member types, for example:

```
func isSorted<S: Sequence>(_: S) where S.Element: Comparable {...}
```

Another generalization over generic parameter inheritance clauses is that `where` clauses can define same-type requirements:

```
func merge<S: Sequence, T: Sequence>(_: S, _: T) -> [S.Element]
    where S: Comparable, S.Element == T.Element {...}
```

Figure 5.1.: Types and requirements, at the syntactic and semantic layers



Formally, a `where` clause is a list of one or more *requirement representations*. There are three kinds of requirement representations, with the first two kinds storing a pair of type representations, and the third storing a type representation and layout constraint:

1. **Constraint requirement representations**, written as `T: C`, where `T` and `C` are type representations, called the subject type and constraint type, respectively.
2. **Same-type requirement representations**, written as `T == U`, where `T` and `U` are type representations.
3. **Layout requirement representations**, written as `T: L` where `L` is a layout constraint. The only type of layout constraint which can be written in the source language is `AnyObject`, but this is actually parsed as a constraint requirement representation. Bona-fide layout requirement representations only appear within the `@_specialize` attribute.

Just as type resolution resolves type representations to types, *requirement resolution* resolves requirement representations to *requirements*. A requirement is the equivalent of a requirement representation at the semantic layer; requirements store types instead of type representations. Figure 5.1 shows the correspondence.

Requirement resolution resolves each type representation to a type, and computes the requirement kind. The requirement kind encodes more detail than the requirement representation kind:

- **Conformance requirements** state that a type must conform to a protocol, protocol composition or parameterized protocol type.
- **Superclass requirements** state that a type must either equal to be a subclass of the superclass type.
- **Layout requirements** state that a type must satisfy a layout constraint.
- **Same-type requirements** state that two interface types are reduced-equal (this concept was first introduced in Chapter 3 and will be detailed in Section 6.3).

Constraint requirement representations resolve to conformance, superclass and layout requirements; the exact kind of requirement is only known after type resolution resolves the constraint type by performing name lookups. Same-type requirement representations always resolve to same-type requirements.

The simpler syntax introduced in the previous section, where a constraint type can be written in the inheritance clause of a generic parameter declaration, also resolves to a requirement. The requirement's subject type is the generic parameter type. The requirement kind is always a conformance, superclass or layout requirement, never a same-type requirement.

History The `where` clause syntax used to be part of the generic parameter list itself, but was moved to the modern "trailing" form in Swift 3 [21]. Implementation limitations prevented `where` clause requirements from constraining outer generic parameters until Swift 3. Once these implementation difficulties were solved, it no longer made sense to restrict a `where` clause to appear only on a declaration that has its own generic parameter list; this restriction was lifted in Swift 5.3 [22], allowing any declaration in a generic context to declare a `where` clause.

For example, the following became valid:

```
enum LinkedList<Element> {
  ...

  func sum() -> Element where Element: AdditiveArithmetic {...}
}
```

There is no semantic distinction between attaching a `where` clause to a member of a type, or moving the member to a constrained extension, so the above is equivalent to the following:

```
extension LinkedList where Element: AdditiveArithmetic {
  func sum() -> Element {...}
}
```

Unfortunately, due to historical quirks in the name mangling scheme, the above is not an ABI-compatible transformation.

Protocol requirements There is still one situation where constraining outer generic parameters is prohibited, for usability reasons. The *value requirements* of a protocol (properties, subscripts and methods) cannot constrain `Self` or its associated types in their `where` clause. The reason is that this value requirements must be fulfilled by all concrete conforming types. If a value requirement's `where` clause imposed additional

constraints on `Self`, it would be impossible for a concrete type which did not otherwise satisfy those constraints to declare a witness for this value requirement. Rather than allow defining a protocol which cannot be conformed to, the type checker diagnoses an error.

Example 5.3. The following protocol attempts to define an `Element` associated type with no requirements, and a `minElement()` method which requires that `Element` conform to the `Comparable` protocol:

```
protocol SetProtocol {
  associatedtype Element

  func minElement() -> Element where Element: Comparable
}
```

This is not allowed, because there is no way to implement the `minElement()` requirement in a concrete conforming type whose `Element` type is not `Comparable`. One way to fix the error is to move the `where` clause from the protocol method to the associated type, which would instead impose the requirement on all conforming types.

5.3. Opaque Parameters

In the type of a function or subscript parameter, the `some` keyword declares an *opaque parameter type*. The `some` keyword is followed by a constraint type. This introduces an unnamed generic parameter, and the constraint type imposes a conformance, superclass or layout requirement on this generic parameter.

If a declaration has both a parsed generic parameter list and opaque parameters, the opaque parameters have the same depth as the parsed generic parameters, and appear after the parsed generic parameters in index order.

Opaque parameter types are unnamed, and therefore are not visible to type resolution. In particular, there is no way to refer to an opaque parameter type within the function's `where` clause, or from a type annotation on a declaration nested in the function's body. From expression context however, the type of an opaque parameter can be obtained via the built-in `type(of:)` pseudo-function,[1] which produces a metatype value. This allows for invoking static methods and such.

Example 5.4. These two definitions are equivalent:

```
func merge<E>(_: some Sequence<E>, _: some Sequence<E>) -> [E] {}
func merge<E, S: Sequence<E>, T: Sequence<E>>(_: S, _: T) -> [E] {}
```

[1] It looks like a function call, but the type checking behavior of `type(of:)` cannot be described by a Swift function type; it is not a real function.

Listing 5.3.: A protocol declaration with a primary associated type which is then used as a parameterized protocol type

```
protocol IteratorProtocol<Element> {
  associatedtype Element
  mutating func next() -> Element?
}

// The first declaration is equivalent to the second:
func sumOfSquares<I: IteratorProtocol<Int>>(_: I) -> Int {...}

func sumOfSquares<I: IteratorProtocol>(_: I) -> Int
    where I.Element == Int {...}
```

The constraint types here are parameterized protocol types, which are described in the next section.

Opaque parameter declarations were introduced in Swift 5.7 [23]. Note that `some` appearing in the return type of a function declares an *opaque return type*, which is a related but quite different feature (Chapter 14).

5.4. Protocol Declarations

Protocols have an implicit generic parameter list with a single generic parameter named `Self`. Conceptually the `Self` type stands in for the concrete conforming type.

Protocols cannot be nested inside any declaration context other than a source file; structs, classes and enums cannot be nested inside of protocols. This restriction is discussed in Section 7.4.

Protocols can specify generic requirements on the `Self` type and its associated types, using similar syntax to other generic declarations. The type checker ensures that these requirements are satisfied by any concrete type conforming to the protocol.

Primary associated types A protocol can declare a list of *primary associated types* with a syntax resembling a generic parameter list. While generic parameter lists introduce new generic parameter declarations, the entries in the primary associated type list reference existing associated types declared in the protocol's body. A protocol with primary associated types can be used as a parameterized protocol type.

As a constraint type, a parameterized protocol type is equivalent to a conformance requirement between the subject type and the protocol, together with one or more same-type requirements. The same-type requirements relate the primary associated types of the subject type with the arguments of the parameterized protocol type.

Example 5.5. Listing 5.3 shows the standard library's iterator protocol, which defines a single primary associated type, together with a use of the protocol as a parameterized protocol type.

Parameterized protocol types and primary associated types were added to the language in Swift 5.7 [15]. This *desugaring* will receive a more formal treatment in Section 11.2.

Associated type requirements Associated types can state one or more constraint types in their inheritance clause, in addition to an optional `where` clause. Constraint types in the inheritance clause resolve to requirements whose subject type is the associated type declaration's declared interface type—which you might recall is the dependent member type `Self.[P]A`. where `A` is an associated type declaration in some protocol `P`. The standard library `Sequence` protocol demonstrates all of these features:

```
protocol Sequence<Element> {
  associatedtype Iterator: IteratorProtocol
  associatedtype Element where Iterator.Element == Element

  func makeIterator() -> Iterator
}
```

The conformance requirement on `Iterator` could have been written with a `where` clause as well:

```
associatedtype Iterator where Iterator: IteratorProtocol
```

Finally, a `where` clause can be attached to the protocol itself; there is no semantic difference between that and attaching it to an associated type:

```
protocol Sequence where Iterator: IteratorProtocol,
                        Iterator.Element == Element {...}
```

Unlike generic parameters, associated type inheritance clauses allow multiple entries, separated by commas. This is effectively equivalent to a single inheritance clause entry containing a protocol composition:

```
associatedtype Data: Codable & Hashable
associatedtype Data: Codable, Hashable
```

Unqualified lookup inside protocols Within the entire source range of the protocol declaration, unqualified references to associated types, like `Element` and `Iterator` above, resolve to their declared interface type. This is a shorthand for accessing the associated type as a member type of the protocol `Self` type. The `Sequence` protocol above could instead have been declared as follows:

```
protocol Sequence where Self.Iterator: IteratorProtocol,
                        Self.Iterator.Element == Self.Element {...}
```

Protocol inheritance clauses Constraint types appearing in the protocol's inheritance clause become generic requirements on `Self` in the same manner that constraint types in generic parameter inheritance clauses become requirements on the generic parameter type. Requirements on `Self` are imposed by the conformance checker on concrete types conforming to the protocol.

If the constraint type is another protocol, we call the protocol stating the requirement the *derived protocol* and the protocol named by the constraint type the *base protocol*. The derived protocol is said to *inherit* from (or sometimes, *refine*) the base protocol. Protocol inheritance can be observed in two ways; first, every concrete type conforming to the derived protocol must also conform to the base protocol. Second, qualified name lookup will search through inherited protocols when the lookup begins from the derived protocol or one of its concrete conforming types.

For example, the standard library's `Collection` protocol inherits from `Sequence`, therefore any concrete type conforming `Collection` must also conform to `Sequence`. If some type parameter `T` is known to conform to `Collection`, members of both the `Collection` and `Sequence` protocols will be visible to qualified name lookup on a value of type `T`.

```
protocol Collection: Sequence {...}
```

Protocols can restrict their conforming types to those with a reference-counted pointer representation by stating an `AnyObject` layout constraint:

```
protocol BoxProtocol: AnyObject {...}
```

Protocols can also impose a superclass requirement on their conforming types:

```
class Plant {}
class Animal {}
protocol Duck: Animal {}
class MockDuck: Plant, Duck {}
// error: MockDuck is not a subclass of Animal
```

Just like with protocol inheritance, qualified name lookup understands a superclass in a protocol's inheritance clause, making the members of the superclass visible to all lookups that look into the protocol.

A protocol is *class-constrained* if the `Self: AnyObject` requirement can be proven from its inheritance clause; either directly stated, implied by a superclass requirement, or inherited from another protocol.

History In older releases of Swift, protocols could only constrain associated types by writing a constraint type in the associated type's inheritance clause, which limited the kinds of requirements that could be imposed on the concrete conforming type. The general trailing `where` clause syntax on associated types and protocols were introduced in Swift 4 [24].

Another important generalization was allowing an associated type to conform to the same protocol that it appears in, either directly or indirectly. For example, the SwiftUI `View` protocol has a `Body` associated type that itself conforms to `View`:

```
protocol View {
  associatedtype Body: View

  var body: Body { get }
}
```

The ability to declare a so-called *recursive conformance* was introduced in Swift 4.1 [25]. This feature has some profound implications. In particular, it means that a generic signature with a conformance to a protocol such as the above has an infinite number of type parameters; for example, consider `<T where T: View>`:

```
T
T.Body
T.Body.Body
T.Body.Body.Body
...
```

5.5. Source Code Reference

Key source files:

- include/swift/AST/Decl.h
- include/swift/AST/DeclContext.h

- include/swift/AST/GenericParamList.h
- lib/AST/Decl.cpp
- lib/AST/DeclContext.cpp
- lib/AST/GenericParamList.cpp

Other source files:

- include/swift/AST/Types.h
- lib/AST/NameLookup.cpp

`DeclContext` *class*

See also Section 2.1, Section 4.4 and Section 6.5.

- `isGenericContext()` answers if this declaration context or one of its parent contexts has a generic parameter list.
- `isInnermostContextGeneric()` answers if this declaration context is a generic context with its own generic parameter list, that is, if its declaration is a generic declaration.

`GenericContext` *class*

Base class for declarations which can be generic. See also Section 6.5.

- `getParsedGenericParams()` returns the declaration's parsed generic parameter list, or `nullptr`.
- `getGenericParams()` returns the declaration's full generic parameter list, which includes any implicit generic parameters. Evaluates a `GenericParamListRequest`.
- `isGeneric()` answers if this declaration has a generic parameter list.
- `getGenericContextDepth()` returns the depth of the innermost generic parameter list, or `(unsigned)-1` if neither this declaration nor any outer declaration is generic.
- `getTrailingWhereClause()` returns the trailing `where` clause, or `nullptr`.

Trailing `where` clauses are not preserved in serialized generic contexts. Except when actually building the generic signature, most code uses `getGenericSignature()` from Section 6.5 instead.

`GenericParamList` *class*

A generic parameter list.

- `getParams()` returns an array of generic parameter declarations.
- `getOuterParameters()` returns the outer generic parameter list, linking multiple generic parameter lists for the same generic context. Only used for extensions of nested generic types.

`GenericParamListRequest` *class*

This request creates the full generic parameter list for a declaration. Kicked off from `GenericContext::getGenericParams()`.

- For protocols, this creates the implicit `Self` parameter.
- For functions and subscripts, calls `createOpaqueParameterGenericParams()` to walk the formal parameter list and look for `OpaqueTypeRepr`s.
- For extensions, calls `createExtensionGenericParams()` which clones the generic parameter lists of the extended nominal itself and all of its outer generic contexts, and links them together via `GenericParamList::getOuterParameters()`.

`GenericTypeParamDecl` *class*

A generic parameter declaration.

- `getDepth()` returns the depth of the generic parameter declaration.
- `getIndex()` returns the index of the generic parameter declaration.
- `getName()` returns the name of the generic parameter declaration.
- `getDeclaredInterfaceType()` returns the non-canonical generic parameter type for this declaration.
- `isOpaque()` answers if this generic parameter is associated with an opaque parameter.
- `getOpaqueTypeRepr()` returns the associated `OpaqueReturnTypeRepr` if this is an opaque parameter, otherwise `nullptr`.
- `getInherited()` returns the generic parameter declaration's inheritance clause.

Inheritance clauses are not preserved in serialized generic parameter declarations. Requirements stated on generic parameter declarations are part of the corresponding generic context's generic signature, so except when actually building the generic signature, most code uses `getGenericSignature()` on Section 6.5 instead.

`GenericTypeParamType` *class*

A generic parameter type.

- `getDepth()` returns the depth of the generic parameter declaration.
- `getIndex()` returns the index of the generic parameter declaration.
- `getName()` returns the name of the generic parameter declaration, only if this is a non-canonical type.

`TrailingWhereClause` *class*

The syntactic representation of a trailing `where` clause.

- `getRequirements()` returns an array of `RequirementRepr`.

`RequirementRepr` *class*

The syntactic representation of a requirement in a trailing `where` clause.

- `getKind()` returns a `RequirementReprKind`.
- `getFirstTypeRepr()` returns the first `TypeRepr` of a same-type requirement.
- `getSecondTypeRepr()` returns the second `TypeRepr` of a same-type requirement.
- `getSubjectTypeRepr()` returns the first `TypeRepr` of a constraint or layout requirement.
- `getConstraintTypeRepr()` returns the second `TypeRepr` of a constraint requirement.
- `getLayoutConstraint()` returns the layout constraint of a layout requirement.

`RequirementReprKind` *enum class*

- `RequirementRepr::TypeConstraint`
- `RequirementRepr::SameType`
- `RequirementRepr::LayoutConstraint`

`WhereClauseOwner` *class*

Represents a reference to some set of requirement representations which can be resolved to requirements, for example a trailing `where` clause. This is used by various requests, such as the `RequirementRequest` below, and the `InferredGenericSignatureRequest` in Section 11.4.

- `getRequirements()` returns an array of `RequirementRepr`.
- `visitRequirements()` resolves each requirement representation and invokes a callback with the `RequirementRepr` and resolved `Requirement`.

`RequirementRequest` *class*

Request which can be evaluated to resolve a single requirement representation in a `WhereClauseOwner`. Used by `WhereClauseOwner::visitRequirements()`.

`ProtocolDecl` *class*

A protocol declaration.

- `getTrailingWhereClause()` returns the protocol `where` clause, or `nullptr`.
- `getAssociatedTypes()` returns an array of all associated type declarations in the protocol.
- `getPrimaryAssociatedTypes()` returns an array of all primary associated type declarations in the protocol.
- `getInherited()` returns the parsed inheritance clause.

Trailing `where` clauses and inheritance clauses are not preserved in serialized protocol declarations. Except when actually building the requirement signature, most code uses `getRequirementSignature()` from Section 6.5 instead.

The last four utility methods operate on the requirement signature, so are safe to use on deserialized protocols:

- `getInheritedProtocols()` returns an array of all protocols directly inherited by this protocol, computed from the inheritance clause.
- `inheritsFrom()` determines if this protocol inherits from the given protocol, possibly transitively.
- `getSuperclass()` returns the protocol's superclass type.
- `getSuperclassDecl()` returns the protocol's superclass declaration.

`AssociatedTypeDecl` *class*

An associated type declaration.

- `getTrailingWhereClause()` returns the associated type's trailing `where` clause, or `nullptr`.

- `getInherited()` returns the associated type's inheritance clause.

Trailing `where` clauses and inheritance clauses are not preserved in serialized associated type declarations. Requirements on associated types are part of a protocol's requirement signature, so except when actually building the requirement signature, most code uses `getRequirementSignature()` from Section 6.5 instead.

6. Generic Signatures

We've now seen all the syntactic building blocks that go into constructing the *generic signature* of a generic context. Each level of generic context nesting can introduce new generic parameters or requirements, so the generic signature collects information from each outer generic declaration. This records in one place a complete description of a generic context:

- A list of all visible generic parameters, including outer parameters. This includes generic parameters explicitly defined in source, as well as those generic parameters implicitly introduced by opaque parameter declarations.
- A list of all generic requirements that apply to these generic parameters, which includes those from outer declarations. We've seen three syntactic forms that define requirements so far: generic parameter inheritance clauses, trailing `where` clauses, and opaque parameters. A fourth and final mechanism, requirement inference, is described later in Section 11.1.

The `-debug-generic-signatures` frontend flag prints the generic signature of each declaration as it is being type checked. In debug output, the printed representation of a generic signature resembles the language syntax; we're going to use this written notation throughout when talking about generic signatures:

```
<A, B, C, ...> where A: P, B == A.[P]T, ...>
 generic parameters          requirements
```

Example 6.1. Listing 6.1 shows three generic declarations and the compiler output from the `-debug-generic-signatures` flag.

The requirements in a generic signature are constructed from syntactic representations, but they do not always look like the requirements written by the user. The requirements in a generic signature satisfy certain invariants and are sorted by comparing their subject types. The type parameter order is introduced in Section 6.2. The multi-step process for transforming user-written requirements into the correct minimal form that appears in a generic signature is described in Chapter 11. For now though, we're just going to assume you're working with an existing generic signature that was given to you by the type checker or some other part of the compiler.

Listing 6.1.: Example program and `-debug-generic-signatures` output

```
struct Outer<T: Sequence> {
  struct Inner<U> {
    func transform() -> (T, U) where T.Element == U {
      ...
    }
  }
}
```

```
debug.(file).Outer@debug.swift:1:8
Generic signature: <T where T : Sequence>

debug.(file).Outer.Inner@debug.swift:2:10
Generic signature: <T, U where T : Sequence>

debug.(file).Outer.Inner.transform()@debug.swift:3:10
Generic signature: <T, U where T : Sequence, U == T.[Sequence]Element>
```

Canonical signatures Generic signatures are immutable and uniqued, so two generic signatures with the same structure and the same sugared types are pointer-equal. A generic signature is *canonical* if all listed generic parameter types are canonical, and any types appearing in requirements are canonical. A canonical signature is computed from an arbitrary generic signature by replacing any sugared types appearing in the signature with canonical types. Two generic signatures are canonical-equal if their canonical signatures are pointer-equal.

Example 6.2. These two declarations state their requirements in different ways, and you might even spot that the second one has a redundant requirement:

```
func allEqual1<T: Sequence<U.Element>, U: Sequence> -> Bool {}

func allEqual2<A, B>(_: A, _: B) -> Bool
    where A: Sequence,
          B: Sequence,
          B.Element == A.Element,
          A.Iterator: IteratorProtocol {}
```

The requirements of both `allEqual1()` and `allEqual2()` reduce to the same form in their generic signatures.

The first declaration's generic signature:

```
<T, U where T: Sequence, U: Sequence,
            T.[Sequence]Element == U.[Sequence]Element>
```

The second declaration's generic signature:

```
<A, B where A: Sequence, B: Sequence,
            A.[Sequence]Element == B.[Sequence]Element>
```

The two generic signatures only differ by type sugar; namely, they use the corresponding sugared generic parameter types from their declaration. This means they are not pointer-equal, but they are canonical-equal. The canonical generic signature of both is obtained by replacing generic parameters with their canonical types:

```
<τ_0_0, τ_0_1
    where τ_0_0: Sequence, τ_0_1: Sequence,
          τ_0_0.[Sequence]Element == τ_0_1.[Sequence]Element>
```

Reduced signatures? There is no notion of a "reduced generic signature" the way we have reduced types. The generic requirements in a generic signature are always written in a minimal, reduced form (Section 11.3); the only variation allowed is type sugar.

Empty generic signature If a nominal type declaration is not a generic context (that is, neither it nor any parent context has any generic parameters), then its generic signature will have no generic parameters or generic requirements. This is called the *empty generic signature*. Lacking any generic parameters, the empty generic signature more generally has no type parameters, either. The valid interface types of the empty generic signature are the fully concrete types, that is, types that do not contain any type parameters.

6.1. Requirement Signatures

The generic signature of a protocol `P` always has a single generic parameter `Self` together with a single conformance requirement `Self: P`:

```
<Self where Self: P>
```

The structure "inside" the `Self` type is described by the *requirement signature* of the protocol. The requirement signature is constructed by collecting requirements from the protocol's inheritance clause, associated type inheritance clauses, and `where` clauses on the protocol's associated types and the protocol itself. Just like with generic signatures, the requirements in a requirement signature are always converted into a *minimal* and *reduced* form.

The `-Xfrontend -debug-generic-signatures` flag prints the requirement signature of each protocol that is type checked. The written representation of a requirement signature looks like a generic signature over the protocol's single `Self` generic parameter.

For example, the requirement signature of the `Sequence` protocol is the following:

```
<Self where Self.Iterator: IteratorProtocol,
            Self.Element == Self.Iterator.Element>
```

Requirement signatures also store a compact description of all protocol type aliases defined within the protocol; these are used when resolving `where` clause requirements involving subject types that name protocol type aliases. Protocol type aliases are not shown by the `-Xfrontend -debug-generic-signatures` flag.

Conformance checking When checking a conformance to a protocol, the type checker must ensure the concrete type satisfies all requirements in the requirement signature:

1. The concrete type must conform to any inherited protocols, which are encoded as conformance requirements on the `Self` type.

2. The concrete type must be a class if the requirement signature imposes a superclass or `AnyObject` requirement on `Self`.

3. Finally, the type witnesses must satisfy any requirements imposed on them by the protocol.

All of the above are instances of the more general problem of checking whether concrete types satisfy generic requirements (Section 10.2). The concrete type must also declare a *type witness* for each of the protocol's associated types (Section 8.3).

The conformance requirements of a protocol's requirement signature are known as *associated conformance requirements* and each corresponding conformance is an *associated conformance* (Section 8.5).

A mildly interesting observation The printed representation of a requirement signature is almost never going to form a valid *generic* signature. The requirement signature of `Sequence` as shown above does not state a conformance requirement `Self: Sequence`, so it does not make sense to talk about requirements involving the `Iterator` and `Element` member types of `Self`. This is certainly a valid requirement signature though, because the `Sequence` protocol does not inherit from itself. If we try to build a generic signature from a requirement signature by adding a conformance requirement to `Self`, then *all other* requirements in the requirement signature will become redundant; they are, after all, implied by the conformance of `Self` to the protocol.

Listing 6.2.: Example showing non-obvious protocol inheritance relationship

```
protocol Base {
  associatedtype Other: Base

  typealias Salary = Int
}

protocol Good: Base {
  typealias Income = Salary
}

// warning: protocol ‘Bad’ should be declared to refine ‘Base’ due to a
// same-type constraint on ‘Self’
protocol Bad {
  associatedtype Tricky: Base where Tricky.Other == Self

  typealias Income = Salary
  // error: cannot find type ‘Salary’ in scope
}
```

Protocol inheritance clauses Recall from Section 5.4 that a constraint type written in a protocol's inheritance clause is equivalent to a `where` clause requirement with a subject type of `Self`, just like a constraint type in a generic parameter's inheritance clause is equivalent to a `where` clause requirement with the generic parameter type as the subject type. This correspondence comes with an important caveat, though.

Qualified name lookup into a protocol type must also look into inherited protocols and the protocol's superclass type, if there is one. However, name lookup can only look directly at syntactic constructs, because in the compiler implementation, name lookup is "downstream" of generics. Building a protocol's requirement signature performs type resolution, which queries name lookup; those name lookups cannot in turn depend on the requirement signature having already been constructed.

For this reason, any `where` clause requirements which introduce protocol inheritance relationships must be written with a subject type of exactly `Self` for qualified name lookup to "understand" them. Protocol inheritance implied by some combination of same-type requirements is not allowed. After a protocol's requirement signature has been built, conformance requirements on `Self` are compared against the protocol's inheritance clause; any unexpected conformance requirements are diagnosed with a warning.

Example 6.3. In Listing 6.2, the `Self` type of the `Bad` protocol is equivalent to the

type parameter `Self.Tricky.Other` via a same-type requirement.

The `Tricky` associated type conforms to `Base`, and the `Other` associated type of `Base` also conforms to `Base`. For this reason, the `Self` type of `Bad` actually conforms to `Base`. However, this inheritance relationship is invisible to name lookup, so resolution of the underlying type of `Income` fails to find the declaration of `Salary`.

After building the protocol's requirement signature, the type checker discovers the unexpected conformance requirement on `Self`, but at this stage, it is too late to attempt the failed name lookup again! For this reason, the compiler instead emits a warning suggesting the user change the declaration of the protocol to `protocol Bad: Base`.

6.2. Type Parameter Order

The type parameters of a generic signature are linearly ordered with respect to each other. Let's begin by defining partial orders and linear orders, which are a special kind of partial order.

Definition 6.1. A *partial order* over some set of objects S is a binary relation $<$ satisfying the following:

- For all $a \in S$, $a \not< a$.
- For all a, b, $c \in S$, if $a < b$ and $b < c$, then $a < c$.

Sometimes in the literature, partial orders use the symbol $\leq$ and require that $a \leq a$ for all $a \in S$. This is equivalent to our definition, because you can define that $a < b$ if and only if $a \leq b$ and $a \neq b$.

Definition 6.2. A *linear order* over some set S is a binary relation $<$ that is a partial order with the additional property that for all a, $b \in S$, exactly one of the following holds:

- $a < b$.
- $b < a$.
- $a = b$.

Sometimes, a linear order is called a "total order." Swift programmers will recognize the `Comparable` protocol as abstracting over types that have an intrinsic linear order.

The linear order on type parameter plays an important role in the Swift ABI:

1. The calling convention of a generic function passes witness tables corresponding to protocol conformance requirements. The conformance requirements are ordered by comparing their subject type.

2. Similarly, the in-memory layout of a witness table stores witness tables corresponding to the protocol's associated conformance requirements. The conformance requirements are again ordered by comparing their subject type.

3. The mangled symbol names of generic functions encode their parameter and return types. If those types contain type parameters, the type parameters are reduced. The next section will give the definition of a reduced type parameter as the type parameter that precedes all other type parameters in its equivalence class.

The intuition for this linear order is easiest if you think of a type parameter as a flat list instead of the ordinary recursive representation—that is, a generic parameter followed by zero or more associated type components, instead of recursively applied dependent member types with a root generic parameter at the end. The first element of this list is a generic parameter type, and subsequent elements are identifiers or associated type declarations.

Definition 6.3. The *length* of a type parameter is the length of its flat list representation. The shortest possible type parameter is a generic parameter type, so the length is always greater than or equal to one. The length can also be defined on the recursive representation; a generic parameter type has length one, a dependent member type has length one greater than the length of its base type.

First, we need a way to linearly order generic parameters and associated type declarations. Then, we'll define the linear order on type parameters using this flat list representation.

Definition 6.4. The linear order on two generic parameters τ_d_i and τ_D_I compares the depth, followed by the index:

- If $d < D$, then $\tau\_d\_i < \tau\_D\_I$.
- If $d > D$, then $\tau\_d\_i > \tau\_D\_I$.
- If $d = D$ and $i < I$, then $\tau\_d\_i < \tau\_D\_I$.
- If $d = D$ and $i > I$, then $\tau\_d\_i > \tau\_D\_I$.
- If $d = D$ and $i = I$, then $\tau\_d\_i = \tau\_D\_I$.

The linear order on generic parameters can be more concisely stated as simply the lexicographic order on (depth, index) pairs.

Definition 6.5. A *root associated type* is an associated type defined in a protocol such that no inherited protocol has an associated type with the same name.

Example 6.4. In the following, `Q.A` is *not* a root associated type, because `Q` inherits `P` and `P` also declares an associated type named `A`:

```
protocol P {
  associatedtype A  // root
}

protocol Q : P {
  associatedtype A  // not a root
  associatedtype B  // root
}
```

Definition 6.6. The linear order on protocols is a lexicographic order on fully-qualified protocol names; that is, their module names are compared first, followed by the protocol declaration name if both are defined in the same module.

Example 6.5. Say the `Barn` module defines a `Horse` protocol, and the `Swift` module defines `Collection`. We have `Barn.Horse` $<$ `Swift.Collection`, since `Barn` $<$ `Swift`.

If the `Barn` module also defines a `Saddle` protocol, then `Barn.Horse` $<$ `Barn.Saddle`; both are from the same module, so we compare protocol names, `Horse` $<$ `Saddle`.

Definition 6.7. The linear order on associated type declarations is a lexicographic order on triples, composed from Definitions 6.5 and 6.6:

1. root associated types always precede non-root associated types.
2. two associated types with the same "root-ness" (meaning both are roots or both are non-roots) but from different protocols are compared with the linear protocol order.
3. two associated types with the same "root-ness" and same protocol are compared by name.

If an erroneous protocol declares two associated types with the same name, the source location or any other arbitrary tie breaker can also be used, since invalid code is never ABI.

Finally, we can define the linear order on type parameters. In the literature, this is known as a *shortlex order*.

Definition 6.8 (Linear order on type parameters)**.** When two type parameters differ in length, the one with shorter length precedes the other. For example, we have $\tau\_2\_0 <$ `τ_1_0.Element`.

When two type parameters have the same length, elements are compared pairwise:

Table 6.1.: Type parameters defined by the generic signature in Example 6.6.

Length 1
`T`
`U`
Length 2
`T.[Sequence]Element`
`T.[Sequence]Iterator`
`T.Element`
`T.Iterator`
`U.[Sequence]Element`
`U.[Sequence]Iterator`
`U.Element`
`U.Iterator`
Length 3
`T.[Sequence]Iterator.[IteratorProtocol]Element`
`T.Iterator.Element`
`U.[Sequence]Iterator.[IteratorProtocol]Element`
`U.Iterator.Element`

1. The first pair of elements are always generic parameter types, so they are compared by Definition 6.4.

2. Subsequent pairs are identifiers or associated types. If one is an identifier and the other is an associated type, the associated type declaration precedes the identifier; that is, unbound type parameters "come after" bound type parameters. If both elements are identifiers, they are compared with the lexicographic order on strings. Associated types are compared by Definition 6.7.

Comparison stops at the first index where the two corresponding elements of each type parameter are distinct. The outcome of the final comparison determines the relative order of the two type parameters. If all elements are pairwise equal, the type parameters have the same length and same elements, so must be canonical-equal.

Example 6.6. Table 6.1 shows all type parameters in the following generic signature, written in type parameter order:

```
<T, U where T: Sequence, U: Sequence,
            T.[Sequence]Element == U.[Sequence]Element>
```

A few unbound type parameters are also thrown in the mix to show how they are ordered with respect to the bound type parameters. Notice how type parameters are ordered

Table 6.2.: Equivalence classes defined by the generic signature in Example 6.6

Reduced type parameter	Representatives
`T`	`T`
`U`	`U`

`T.[Sequence]Element`	`T.[Sequence]Element`
	`T.Element`
	`U.[Sequence]Element`
	`U.Element`
	`T.[Sequence]Iterator.[IteratorProtocol]Element`
	`T.Iterator.Element`
	`U.[Sequence]Iterator.[IteratorProtocol]Element`
	`U.Iterator.Element`

`T.[Sequence]Iterator` `T.Iterator`	`T.[Sequence]Iterator` `T.Iterator`
`U.[Sequence]Iterator` `U.Iterator`	`U.[Sequence]Iterator` `U.Iterator`

by length first; all type parameters of length 1 precede those of length 2, which precede those of length 3.

6.3. Reduced Types

Two type parameters are *equivalent* with respect to a generic signature if one can be transformed into the other via a series of same-type requirements. The set of all type parameters equivalent to a given type parameter is called its *equivalence class*. Every type parameter is part of exactly one equivalence class, so the set of all type parameters described by a generic signature can be partitioned into disjoint equivalence classes.

Definition 6.9. A type parameter is a *reduced type parameter* with respect to a generic signature if it is not fixed to a concrete type, and precedes every other type parameter in its own equivalence class. Type parameters fixed to concrete types are never considered to be reduced.

Definition 6.10. An interface type is a *reduced type* with respect to a generic signature if all type parameters appearing inside the interface type are reduced type parameters. It follows that an interface type containing a type parameter that is fixed to a concrete type is not a reduced type.

Example 6.7. Table 6.2 groups the type parameters from Example 6.6 into equivalence classes. The type parameters in the first column are the reduced types of the type parameters in the second column.

The generic parameters `T` and `U` are in their own equivalence class. The equivalence class of `T.[Sequence]Element` contains multiple type parameters, because of the same-type requirement between the element types of `T` and `U`. Then there are two equivalence classes for the iterator types, `T.[Sequence]Iterator` and `U.[Sequence]Iterator`.

The type parameters of an equivalence class are ordered; the first type parameter is the reduced type parameter for all members of that equivalence class. The equivalence classes themselves are also ordered, by comparing their reduced type parameters.

You can think of a same-type requirement as merging two equivalence classes together into a larger equivalence class. The equivalence class `T.[Sequence]Element` was formed by two same-type requirements:

1. `Self.Element == Self.IteratorProtocol.Element`, in the `Sequence` protocol.
2. `T.Element == U.Element`, in our generic signature.

If we omit the second requirement, `T.[Sequence]Element` and `U.[Sequence]Element` would belong to two different equivalence classes. Each equivalence class would still contain the element type of the corresponding iterator, because of the first same-type requirement.

For a generic signature, we can can construct a directed graph called the *equivalence class graph*. A directed graph is defined by a set of vertices, and a set of edges, which are ordered pairs of vertices. The vertices here are reduced type parameters. There is an edge from a type parameter `T` to a type parameter `U` if for some associated type declaration `A` in a protocol `P`, `T` is conforms to `P`, and `T.[P]A` reduces to `U`. Edges are labeled with their associated type declarations.

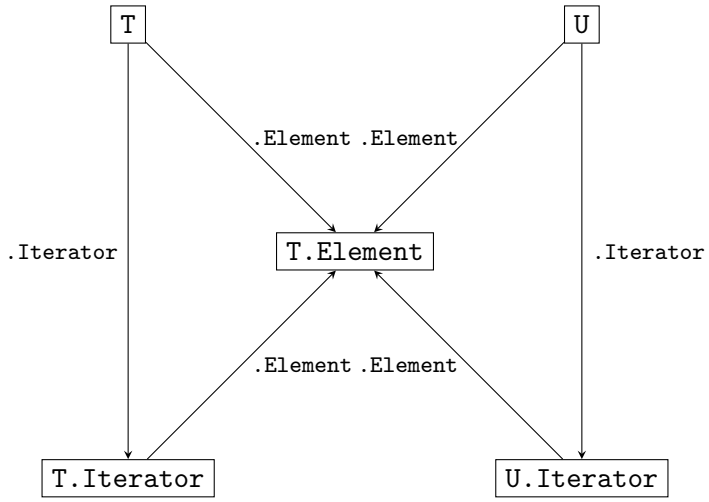
A type parameter can be thought of as a *path* through this directed graph, starting from a generic parameter, then traversing successive edges for each associated type declaration until reaching the type parameter's equivalence class. Two reduced-equal type parameters represent two different paths that end at the same equivalence class.

Example 6.8. Figure 6.1 shows the equivalence class graph for the generic signature of Example 6.6:

- If you begin at the generic parameter `T` and follow the `.Element` edge, you end up at the equivalence class whose reduced type is `T.Element`.
- Similarly, if you begin at the generic parameter `U`, then follow the `.Iterator` edge, and finally follow the `.Element` edge, you also end up at the equivalence class with the reduced type `T.Element`.

This shows that the type parameters `T.Element` and `U.Iterator.Element` belong to the same equivalence class, with the reduced type of `T.Element`.

Figure 6.1.: The directed graph of equivalence classes from Example 9.1



The equivalence class graph is a useful intuition, but it does not yield a useful computational algorithm. The set of equivalence classes may be infinite, as you saw with the SwiftUI `View` protocol shown in Section 5.4. It is also possible for a single equivalence class to consist of an infinite set of type parameters. An example appears in the standard library `Collection` protocol, which has a `SubSequence` associated type:

```
protocol Collection: Sequence {
  ...
  associatedtype SubSequence: Collection
    where SubSequence == SubSequence.SubSequence
  ...
}
```

In the generic signature `<T where T: Collection>`, all of the following type parameters belong to the same equivalence class, via the same-type requirement:

```
T.SubSequence
T.SubSequence.SubSequence
T.SubSequence.SubSequence.SubSequence
...
```

Mathematical aside When we defined reduced type parameters, we assumed that each equivalence class has a unique smallest type parameter. This might seem obvious, but

it is not always true for arbitrary infinite sets and linear orders. For example, the set of negative integers is an infinite set that can be linearly ordered with the standard "less-than" relation, but it does not have a minimum element, because we can exhibit an *infinite descending chain* where each integer is smaller than the next:

$$\cdots < -3 < -2 < -1$$

With the type parameter order, this cannot happen; it is a *well-founded order*. This allows us to reduce the problem of finding the minimum element of an equivalence class to the problem of finding the minimum element of a *finite* set of type parameters, as follows:

1. The set of type parameters of any fixed length N is finite, because there are a finite number of generic parameters and associated type declarations in a program, and each type parameter is obtained by combining them in a finite number of possible ways.

2. Therefore, the set of type parameters of length $\leq N$ is also finite.

3. This means that the set of type parameters that precede some type parameter T of length N under our linear order is always finite, because it is a subset of the set of type parameters of length $\leq N$.

4. Now, we pick an arbitrary type parameter from our equivalence class, and find the finite subset of type parameters that are smaller than or equal to our chosen type parameter. A finite set always has a minimum element; this is the smallest type parameter of our equivalence class.

There is an interesting corollary to the above argument: any infinite equivalence class of type parameters must contain type parameters of arbitrary length.

6.4. Generic Signature Queries

A few times, we've mentioned "proving" properties that are implied by some combination of generic requirements. A fundamental set of *generic signature queries* are used by the rest of the compiler to reason about the type parameters of a generic signature. This section just defines their behavior; a full accounting of how generic signature queries are *implemented* will have to wait until Chapter 18.

The various kinds of queries are grouped into three categories, shown in Table 6.3.

Predicate queries The simplest of all queries are the binary predicates, which respond with `true` or `false`.

Table 6.3.: Generic signature queries

Predicates	`isValidTypeParameter()`
	`requiresProtocol()`
	`requiresClass()`
	`isConcreteType()`

Properties	`getRequiredProtocols()`
	`getSuperclassBound()`
	`getConcreteType()`
	`getLayoutConstraint()`

Reduced types	`areReducedTypeParametersEqual()`
	`isReducedType()`
	`getReducedType()`

`isValidTypeParameter()` answers if a type parameter is valid for this generic signature.

`requiresProtocol()` answers if a type parameter conforms to a protocol.

`requiresClass()` answers if a type parameter is subject to an `AnyObject` layout constraint, meaning it is represented at runtime as a single retainable pointer. This can either be stated explicitly, or implied by a superclass requirement.

`isConcreteType()` answers if a type parameter is fixed to a concrete type.

Example 6.9. Consider this pair of generic signatures:

```
<E where E: Sequence>
<E, F where E: Sequence, E.[Sequence]Element: Sequence,
            F == E.[Sequence]Element.[Sequence]Element>
```

- `isValidTypeParameter(E)` is true in both signatures.
- `isValidTypeParameter(F)` is only true in the second signature, because the first signature only has one generic parameter.
- `isValidTypeParameter(E.Element)` is true in both signatures.
- `isValidTypeParameter(E.Element.Element)` is only true in the second signature, because `E.Element` does not conform to `Sequence` in the first signature.

Example 6.10. Consider this generic signature:

```
<T, U, V where T: Collection, T.[Sequence]Element == Array<U>,
              U: Executor, V: NSObject>
```

The following queries all return true:

- `requiresProtocol(T, Collection)`, because the requirement is directly stated.
- `requiresProtocol(T, Sequence)`, because `Collection` inherits from `Sequence`.
- `requiresProtocol(T.Iterator, IteratorProtocol)`, because the `Iterator` associated type of `Sequence` conforms to `IteratorProtocol`.
- `requiresClass(U)`, because `Executor` is a class-constrained protocol.
- `requiresClass(V)`, because `NSObject` is a class.
- `isConcreteType(T.Element)`, because the requirement is directly stated.
- `isConcreteType(T.Iterator.Element)`, implied by the same-type requirement in the requirement signature of `Sequence`.

Property queries The next set of queries derive more complex properties that are not just true/false predicates.

`getRequiredProtocols()` returns the list of all protocols that a type parameter must conform to. The list is minimal in the sense that no protocol inherits from any other protocol in the list, and sorted in canonical protocol order (Definition 6.6).

`getSuperclassBound()` returns the superclass bound of a type parameter if there is one.

`getConcreteType()` returns the concrete type to which a type parameter is fixed if there is one.

`getLayoutConstraint()` returns the layout constraint describing a type parameter's runtime representation if there is one.

The `AnyObject` layout constraint is the only one that can be explicitly written in source. A second kind of layout constraint, `_NativeClass`, is implied by a superclass requirement whose superclass is a native Swift class, meaning a class not inheriting from `NSObject`. The `_NativeClass` layout constraint implies the `AnyObject` layout constraint.

The two differ in how reference counting operations on their instances are lowered in code generation; arbitrary class instances use the Objective-C runtime entry points for retain and release operations, whereas native class instances use a more efficient calling convention.

Example 6.11. In the following generic signature, `getSuperclassBound(T)` is `G<U>`:

```
<T, U where T: G<U>>

class G<A> {}
```

Example 6.12. In the following generic signature, `getConcreteType(T.Index)` is `Int`:

```
<T where T: Collection, T.[Collection]Indices == Range<Int>>
```

This is a non-trivial consequence of several requirements:

- The type parameter `T.[Collection]Index` is in the equivalence class of the type parameter `T.[Collection]Indices.[Sequence]Element`, via the same-type requirement in the `Collection` protocol.
- The base type of this type parameter is `T.[Collection]Indices`, which is fixed to the concrete type `Range<Int>` in our generic signature.
- Therefore, any member types of this type parameter are fixed to the corresponding type witnesses in the concrete type's conformance.
- The standard library defines a conditional conformance of `Range` to `Collection` when the `Element` generic parameter of `Range` conforms to the `Strideable` protocol:

  ```
  extension Range: Collection where Element: Strideable {...}
  ```

  Since `Int` conforms to `Strideable`, the type `Range<Int>` satisfies the conditional requirements of this conditional conformance.

  The `Element` associated type is witnessed by the `Element` generic parameter in the conformance of `Range<Element>` to `Sequence`.
- The type parameter `T.[Collection]Indices.[Sequence]Element` is therefore fixed to the concrete type `Int`, which gives us the final result.

Reduced type queries The final three generic signature queries concern reduced types:

`areReducedTypeParametersEqual()` answers if two type parameters have the reduced type. Does not produce a useful result if one or the other is concrete.

`isReducedType()` answers if an arbitrary type is already reduced.

`getReducedType()` computes the reduced type of an arbitrary type.

Example 6.13. In the generic signature `<T where T: Sequence>`, the reduced type of `Array<T.Iterator.Element>` is `Array<T.Element>`.

6.5. Source Code Reference

Key source files:

- include/swift/AST/GenericSignature.h
- include/swift/AST/Requirement.h
- include/swift/AST/RequirementSignature.h
- lib/AST/GenericSignature.cpp

Other source files:

- include/swift/AST/Decl.h
- include/swift/AST/DeclContext.h
- lib/AST/Decl.cpp
- lib/AST/DeclContext.cpp

`DeclContext` *class*

See also Section 4.4 and Section 5.5.

- `getGenericSignatureOfContext()` returns the generic signature of the innermost generic context, or the empty generic signature if there isn't one.

`GenericContext` *class*

See also Section 5.5.

- `getGenericSignature()` returns the declaration's generic signature, computing it first if necessary. If the declaration does not have a generic parameter list or trailing `where` clause, returns the generic signature of the parent context.

`GenericSignature` *class*

Represents an immutable, uniqued generic signature. Meant to be passed as a value, it stores a single instance variable, a `GenericSignatureImpl *` pointer.

The `getPointer()` method returns this pointer. The pointer is not `const`, however `GenericSignatureImpl` does not define any mutating methods.

The pointer may be `nullptr`, representing an empty generic signature; the default constructor `GenericSignature()` constructs this value. There is an implicit `bool` conversion which tests for the empty generic signature.

The `getPointer()` method is only used occasionally, because the `GenericSignature` class overloads `operator->` to forward method calls to the `GenericSignatureImpl *` pointer. Some operations on generic signatures are methods on `GenericSignature` (called with "`.`") and some on `GenericSignatureImpl` (called with "`->`").

Methods of `GenericSignature` are safe to call with an empty generic signature, which is presented as having no generic parameters or requirements. Methods forwarded to `GenericSignatureImpl` can only be invoked if the signature is non-empty.

The `GenericSignature` class explicitly deletes `operator==` and `operator!=` to make the choice between pointer and canonical equality explicit. To check pointer equality of generic signatures, first unwrap both sides with a `getPointer()` call:

```
1 if (lhsSig.getPointer() == rhsSig.getPointer())
2   ...;
```

The more common canonical signature equality check is implemented by the `isEqual()` method on `GenericSignatureImpl`:

```
1 if (lhsSig->isEqual(rhsSig))
2   ...;
```

Various accessor methods:

- `getGenericParams()` returns an array of `GenericTypeParamType`. If the generic signature is empty, this is the empty array, otherwise it contains at least one generic parameter.
- `getInnermostGenericParams()` returns an array of `GenericTypeParamType` with the innermost generic parameters only, that is, those with the highest depth. If the generic signature is empty, this is the empty array, otherwise it contains at least one generic parameter.
- `getRequirements()` returns an array of `Requirement`. If the generic signature is empty, this is the empty array.
- `getCanonicalSignature()` returns the canonical signature. If the generic signature is empty, returns the canonical empty generic signature.
- `getPointer()` returns the underlying `GenericSignatureImpl *`.

Computing reduced types:

- `getReducedType()` returns the reduced type of an interface type for this generic signature. If the generic signature is empty, the type must be fully concrete, and is returned unchanged.

Other:

- `print()` prints the generic signature, with various options to control the output.
- `dump()` prints the generic signature, meant for use from the debugger or ad-hoc print debug statements.

`GenericSignatureImpl` *class*

The backing storage of a generic signature. Instances of this class are allocated in the AST context, and are always passed by pointer.

- `isEqual()` checks if two generic signatures are canonically equal.
- `getSugaredType()` given a type containing canonical type parameters that is understood to be written with respect to this generic signature, replaces the generic parameter types with their "sugared" forms, so that the name is preserved when the type is printed out to a string.
- `forEachParam()` invokes a callback on each generic parameter of the signature; the callback also receives a boolean indicating if the generic parameter type is reduced or not—a generic parameter on the left hand side of a same-type requirement is not reduced.
- `areAllParamsConcrete()` answers if all generic parameters are fixed to concrete types via same-type requirements, which makes the generic signature somewhat like an empty generic signature. Fully-concrete generic signatures are lowered away at the SIL level.

The generic signature queries from Section 6.4 are methods on `GenericSignatureImpl`:

- Predicate queries:
  - `isValidTypeParameter()`
  - `requiresProtocol()`
  - `requiresClass()`
  - `isConcreteType()`
- Property queries:
  - `getRequiredProtocols()`
  - `getSuperclassBound()`
  - `getConcreteType()`

  - `getLayoutConstraint()`

- Reduced type queries:
  - `areReducedTypeParametersEqual()`
  - `isReducedType()`
  - `getReducedType()`

`CanGenericSignature` *class*

The `CanGenericSignature` class wraps a `GenericSignatureImpl *` pointer which is known to be canonical. The pointer can be recovered with the `getPointer()` method. There is an implicit conversion from `CanGenenericSiganture` to `GenericSignature`. The `operator->` forwards method calls to the underlying `GenericSignatureImpl`.

The `operator==` and `operator!=` operators are used to test `CanGenericSignature` for pointer equality. The `isEqual()` method of `GenericSignatureImpl` implements canonical equality on arbitrary generic signatures by first canonicalizing both sides, then checking the resulting canonical signatures for pointer equality. Therefore, the following are equivalent:

```
1 if (lhsSig->isEqual(rhsSig))
2   ...;
3
4 if (lhsSig.getCanonicalSignature() == rhsSig.getCanonicalSignature())
5   ...;
```

The `CanGenericSignature` class inherits from `GenericSignature`, and so inherits all of the same methods. Additionally, it overrides `getGenericParams()` to return an array of `CanGenericTypeParamType`.

`Requirement` *class*

A generic requirement.

- `getKind()` returns the `RequirementKind`.
- `getSubjectType()` returns the subject type.
- `getConstraintType()` returns the constraint type if the requirement kind is not `RequirementKind::Layout`, otherwise asserts.
- `getProtocolDecl()` returns the protocol declaration of the constraint type if this is a conformance requirement with a protocol type as the constraint type.

- `getLayoutConstraint()` returns the layout constraint if the requirement kind is `RequirementKind::Layout`, otherwise asserts.

`RequirementKind` *enum class*

An enum encoding the four kinds of requirements.

- `RequirementKind::Conformance`
- `RequirementKind::Superclass`
- `RequirementKind::Layout`
- `RequirementKind::SameType`

`ProtocolDecl` *class*

See also Section 5.5.

- `getRequirementSignature()` returns the protocol's requirement signature, first computing it, if necessary.

`RequirementSignature` *class*

A protocol requirement signature.

- `getRequirements()` returns an array of `Requirement`.
- `getTypeAliases()` returns an array of `ProtocolTypeAlias`.

`ProtocolTypeAlias` *class*

A protocol type alias descriptor.

- `getName()` returns the name of the alias.
- `getUnderlyingType()` returns the underlying type of the type alias. This is a type written in terms of the type parameters of the requirement signature.

`TypeBase` *class*

See also Section 3.6.

- `isTypeParameter()` answers if this type is a type parameter; that is, a generic parameter type, or a `DependentMemberType` whose base is another type parameter.

- `hasTypeParameter()` answers if this type is itself a type parameter, or if it contains a type parameter in structural position. For example, `Array<τ_0_0>` will answer `false` to `isTypeParameter()`, but `true` to `hasTypeParameter()`.

`DependentMemberType` *class*

A type representing a reference to an associated type.

- `getBase()` returns the base type; for example, given `τ_0_0.Foo.Bar`, will answer `τ_0_0.Foo`.
- `getName()` returns the identifier naming the associated type.
- `getAssocType()` if this is a resolved `DependentMemberType`, returns the associated type declaration, otherwise if it is unresolved, returns `nullptr`.

`TypeDecl` *class*

See also Section 4.4.

- `compare()` compares two protocols by the protocol order (Definition 6.6), returning one of the following:
  - -1 if this protocol precedes the given protocol,
  - 0 if both protocol declarations are equal,
  - 1 if this protocol follows the given protocol.

`swift::compareDependentTypes()` *function*

Implements the type parameter order (Definition 6.8), returning one of the following:

- -1 if the left hand side precedes the right hand side,
- 0 if the two type parameters are equal as canonical types,
- 1 if the left hand side follows the right hand side.

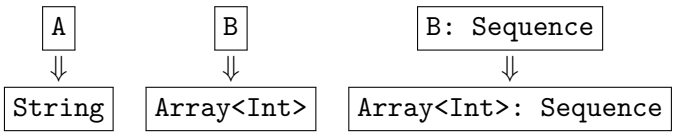
7. Substitution Maps

A *substitution map* describes a mapping from type parameters of a generic signature to replacement types which satisfy the requirements of this generic signature. Substitution maps arise when a reference to a generic declaration is *specialized* by applying generic arguments.

The generic signature of a substitution map is called the *input generic signature*. A substitution map stores a reference to its input generic signature, and the list of generic parameters and conformance requirements in this signature determine the substitution map's shape:

```
<A, B where B: Sequence, B.[Sequence]Element == Int>
```

A substitution map consists of a replacement type for each generic parameter, and a conformance for each conformance requirement:



We can collect all of the above information in a table:

Generic parameters		**Types**
`A`	⇒	`String`
`B`	⇒	`Array<Int>`
Requirements		**Conformances**
`B: Sequence`	⇒	`Array<Int>: Sequence`

Or more concisely,

Types
`A := String`
`B := Array<Int>`

Conformances
`Array<Int>: Sequence`

Listing 7.1.: Substitution maps in type checking

```
1  func genericFunction<A, B: Sequence>(_: A, _: B)
2      where B.Element == Int {}
3
4  struct GenericType<A, B: Sequence> where B.Element == Int {
5    func nonGenericMethod() {}
6  }
7
8  // substitution map for the call is {A := String, B := Array<Int>}.
9  genericFunction("hello", [1, 2, 3])
10
11 // the type of 'value' is GenericType<String, Array<Int>>.
12 let value = GenericType<String, Array<Int>>()
13
14 // the context substitution map for the type of 'value' is
15 // {A := String, B := Array<Int>}.
16 value.nonGenericMethod()
```

Example 7.1. Listing 7.1 shows how our substitution map arises when type checking some code:

Types		
`A`	:=	`String`
`B`	:=	`Array<Int>`

Conformances
`Array<Int>: Sequence`

Here, all three of `genericFunction()`, `GenericType` and `nonGenericMethod()` have the same generic signature, `<A, B where B: Sequence, B.Element == Int>`. When type checking a generic function call, the expression type checker infers the generic arguments from the types of the argument expressions. When referencing a generic type, the generic arguments can be written explicitly.

All three generic declarations are referenced with the same substitution map in this example. (When referencing a generic type declaration, this substitution map is called the *context substitution map* of the specialized type, which is `GenericType<String, Array<Int>>` here. Context substitution maps are coming right up in Section 7.1.)

Type substitution Applying a substitution map to a generic parameter projects the corresponding replacement type from the substitution map.

A type parameter is not necessarily a generic parameter type; it might be a dependent member type as well. Applying a substitution map to a dependent member type derives the replacement type from one of the substitution map's conformances.

Now, we haven't talked about conformances yet. There is an inherent circularity between substitution maps and conformances—substitution maps store conformances, and conformances can store substitution maps, which means that whichever one you choose to explain first, you necessarily have to hand-wave the existence of the other. We will look at conformances in great detail in Chapter 8. The derivation of replacement types for dependent member types is discussed in Section 8.4.

Recall that an interface type is a type *containing* type parameters valid for some generic signature. A substitution map can be more generally applied to an interface type, not just a type parameter. Called *type substitution*, this operation recursively transforms any type parameters appearing in the interface type with their replacement types, preserving the "concrete structure" of the interface type.

The interface type here is called the *original type*, and the result type the *substituted type*. It can be helpful to think of applying a substitution map to an interface type as a *right action*:

$$\boxed{\text{original type}} \times \boxed{\text{substitution map}} = \boxed{\text{substituted type}}$$

Type substitution does not care about generic parameter sugar in the original type; replacement types for generic parameters are always looked up by depth and index in the substitution map.

Example 7.2. Applying the substitution map from our running example to sugared and canonical generic parameter types produces the same results:

$$\left\{ \begin{array}{l} \boxed{\texttt{A}} \\ \boxed{\texttt{τ\_0\_0}} \\ \boxed{\texttt{B}} \\ \boxed{\texttt{τ\_0\_1}} \end{array} \right\} \times \begin{array}{|l|} \hline \textbf{Types} \\ \hline \texttt{A := String} \\ \texttt{B := Array<Int>} \\ \hline \textbf{Conformances} \\ \hline \texttt{Array<Int>: Sequence} \\ \hline \end{array} = \left\{ \begin{array}{l} \boxed{\texttt{String}} \\ \boxed{\texttt{String}} \\ \boxed{\texttt{Array<Int>}} \\ \boxed{\texttt{Array<Int>}} \end{array} \right\}$$

Example 7.3. Listing 7.2 shows a generic type with four member type alias declarations. There are four global variables, and the type of each global variable is written as a member type alias reference with the same type base type, `GenericType<String, Array<Int>>`.

Type resolution resolves a member type alias reference by applying a substitution map to the underlying type of the type alias declaration. Here, the underlying type of each type alias declaration is an interface type for the generic signature of `GenericType`, and the substitution map is the same substitution map as Example 7.1.

Listing 7.2.: Applying a substitution map to four interface types

```
struct GenericType<A, B: Sequence> where B.Element == Int {
  typealias T1 = A
  typealias T2 = B
  typealias T3 = (A.Type, Float)
  typealias T4 = (Optional<A>) -> B
}

let t1: GenericType<String, Array<Int>>.T1 = ...
let t2: GenericType<String, Array<Int>>.T2 = ...
let t3: GenericType<String, Array<Int>>.T3 = ...
let t4: GenericType<String, Array<Int>>.T4 = ...
```

Applying the substitution map to the underlying type of each type alias declaration yields the type of each global variable:

	Original type	**Substituted type**
T1	A	String
T2	B	Array<Int>
T3	(A.Type, Float)	(String.Type, Float)
T4	(Optional<A>) -> B	(Optional<String>) -> Array<Int>

The first two original types are generic parameters, and substitution directly projects the corresponding replacement type from the substitution map; the second two original types are substituted by recursively replacing generic parameters they contain.

References to generic type alias declarations are more complex because in addition to the generic parameters of the base type, the generic type alias will have generic parameters of its own. Section 10.1 describes how the substitution map is computed in this case.

Substitution can *fail* if the interface type contains member types and some of the conformances in the substitution map are invalid. In this case, an error type is returned instead of signaling an assertion. Invalid conformances can appear in substitution maps when the user's own code is invalid; it is not an invariant violation as long as other errors are diagnosed elsewhere and the compiler does not proceed to SILGen with error types in the abstract syntax tree.

Output generic signature If the replacement types in the substitution map are fully concrete—that is, they do not contain type parameters—then all possible substituted

types produced by this substitution map will always be fully concrete. If the replacement types are interface types for some *output* generic signature, the substitution map's substituted types will be written in terms of the type parameters of the output generic signature. The output generic signature might be a different generic signature than the *input* generic signature of the substitution map. This leads naturally to the concept of substitution map composition, described in Section 7.2.

The output generic signature is not stored in the substitution map; it is implicit from context. Also, fully concrete types can be seen as valid interface types for *any* generic signature, because they do not contain type parameters at all. Keeping that in mind, we have this rule:

> **Substitution maps transform the interface types of an input generic signature into the interface types of an output generic signature.**

We haven't introduced archetypes yet, but substitution maps whose replacement types are archetypes will be discussed in Section 9.1.

Canonical substitution maps Substitution maps are immutable and uniqued, just like types and generic signatures. A substitution map is canonical if all replacement types are canonical types and all conformances are canonical conformances. A substitution map is canonicalized by constructing a new substitution map from the original substitution map's canonicalized replacement types and conformances.

As with types, canonicalization gives substitution maps two levels of equality; two substitution maps are pointer-equal if their replacement types and conformances are pointer-equal. Two substitution maps are canonical-equal if their canonical substitution maps are pointer-equal; or equivalently, if their replacement types and conformances are canonical-equal.

Applying a canonical substitution map to a canonical original type is not guaranteed to produce a canonical substituted type. However, there are two important invariants that do hold:

1. Given two canonical-equal original types, applying the same substitution map to both will produce two canonical-equal substituted types.

2. Given an original type and two canonical-equal substitution maps, applying the two substitution maps to this type will also produce two canonical-equal substituted types.

7.1. Context Substitution Maps

A nominal type is *specialized* if the type itself or one of its parent types is a generic nominal type. That is, `Array<Int>` and `Array<Int>.Iterator` are both specialized

types, but `Int` and `String.UTF8View` are not. Equivalently, a nominal type is specialized if the nominal type declaration is a generic context—that is, the type declaration itself has a generic parameter list, or an outer declaration context has one.

Every specialized type determines a unique substitution map for the generic signature of its declaration, called the *context substitution map*. The context substitution map replaces the generic parameters of the type declaration with the corresponding generic arguments of the specialized type.

The defining property is that applying a specialized type's context substitution map to the declared interface type of the type declaration gives us back the specialized type:

$$\boxed{\text{declared interface type}} \times \boxed{\text{context substitution map}} = \boxed{\text{specialized type}}$$

To demonstrate the above identity, consider the generic signature of the `Dictionary` type declaration in the standard library:

```
<Key, Value where Key: Hashable>
```

One possible specialized type for `Dictionary` is the type `Dictionary<Int, String>`; this type, its context substitution map and the declared interface type of `Dictionary` are related as follows:

$$\boxed{\texttt{Dictionary<}\tau\texttt{\_0\_0, }\tau\texttt{\_0\_1>}} \times \left|\begin{array}{|lcl|}\hline \multicolumn{3}{|l|}{\textbf{Types}} \\ \hline \tau\texttt{\_0\_0} & := & \texttt{Int} \\ \tau\texttt{\_0\_1} & := & \texttt{String} \\ \hline \multicolumn{3}{|l|}{\textbf{Conformances}} \\ \hline \multicolumn{3}{|l|}{\texttt{Int: Hashable}} \\ \hline \end{array}\right| = \boxed{\texttt{Dictionary<Int, String>}}$$

The identity substitution map What about the context substitution map of a type declaration's declared interface type? By definition, this substitution map must leave the declared interface type unchanged. That is, it maps every generic parameter of the type declaration's generic signature to itself. If we look at the `Dictionary` type again, we get

$$\boxed{\texttt{Dictionary<}\tau\texttt{\_0\_0, }\tau\texttt{\_0\_1>}} \times \left|\begin{array}{|lcl|}\hline \multicolumn{3}{|l|}{\textbf{Types}} \\ \hline \tau\texttt{\_0\_0} & := & \tau\texttt{\_0\_0} \\ \tau\texttt{\_0\_1} & := & \tau\texttt{\_0\_1} \\ \hline \multicolumn{3}{|l|}{\textbf{Conformances}} \\ \hline \multicolumn{3}{|l|}{\tau\texttt{\_0\_0: Hashable}} \\ \hline \end{array}\right| = \boxed{\texttt{Dictionary<}\tau\texttt{\_0\_0, }\tau\texttt{\_0\_1>}}$$

Every generic signature has such a substitution map, called the *identity substitution map*.

$$\boxed{\text{interface type}} \times \boxed{\text{identity substitution map}} = \boxed{\text{interface type}}$$

Applying the identity substitution map to any interface type leaves it unchanged, with three caveats:

1. The interface type must only contain type parameters which are valid in the input generic signature of this identity substitution map.
2. Substitution might change type sugar, because generic parameters appearing in the original interface type might be sugared differently than the input generic signature of this identity substitution map. Therefore, canonical equality of types is preserved, not necessarily pointer equality.
3. We won't talk about archetypes until Chapter 9, but you may have met them already. Applying the identity substitution map to a contextual type containing archetypes replaces the archetypes with equivalent type parameters. There is a corresponding *forwarding substitution map* which maps all generic parameters to archetypes; the forwarding substitution map acts as the identity in the world of contextual types.

The empty substitution map The empty generic signature only has a single unique substitution map, the *empty substitution map*, so the context substitution map of a non-specialized nominal type is the empty substitution map.

Recall that the only valid interface types of the empty generic signature are the fully concrete types. The action of the empty substitution map leaves fully concrete types unchanged.

$$\boxed{\text{fully-concrete type}} \times \boxed{\text{empty substitution map}} = \boxed{\text{fully-concrete type}}$$

$$\boxed{\texttt{Int}} \times \boxed{\text{empty substitution map}} = \boxed{\texttt{Int}}$$

In general, the empty substitution map is not the same as the identity substitution map. The empty substitution map is the identity substitution map of the empty generic signature only. Applying the empty substitution map to an interface type containing type parameters is a substitution failure and returns an error type.

$$\boxed{\texttt{τ\_0\_0.[Sequence]Element}} \times \boxed{\text{empty substitution map}} = \boxed{\texttt{<<error type>>}}$$

Other declaration contexts A more general notion is the context substitution map of a type *with respect to a declaration context*. This is where the "context" comes from in "context substitution map." Recall that a qualified name lookup `foo.bar` looks for a member named `foo` on some base type, here the type of `foo`. The context substitution map for the member's declaration context describes the substitutions for computing the type of the member reference expression.

When the declaration context is the type declaration itself, "context substitution map with respect to its own declaration context" coincides with the earlier notion of "the" context substitution map of a base type.

Recall from Section 2.1 that qualified name lookup performs a series of *direct lookups*, first into the type declaration itself, then its superclass if any, and finally any protocols it conforms to. A *direct lookup* in turn searches the immediate members of the type declaration and any of its extensions. Thus we can talk about the set of declaration contexts *reachable* from a qualified name lookup on a base type:

1. The type declaration itself and its extensions.
2. The superclass declaration and its extensions, and everything reachable recursively via the superclass declaration.
3. All protocol conformances of the type declaration, and their protocol extensions.

The declaration context for computing a context substitution map must be reachable via qualified name lookup from the base type.

Definition 7.1. The context substitution map with respect to a declaration context is defined as follows for the three kinds of reachable declaration contexts:

1. When the declaration context is the generic type or an extension, the replacement types of the substitution map are the corresponding generic arguments of the base type. If the context is a constrained extension, the substitution map will store additional conformances for the conformance requirements of the extension.
2. When the declaration context is a protocol or a protocol extension, the generic signature is the protocol generic signature, possibly with additional requirements if the context is a constrained protocol extension. The substitution map's single replacement type is the entire base type.
3. When the declaration context is a superclass of the generic type (which must be a class type or an archetype with a superclass requirement), the context substitution map is constructed recursively from the type declaration's superclass type. This case will be described in Chapter 16.

The context substitution map's input generic signature is the generic signature of the declaration context; thus it can be applied to the interface type of a member of this context.

Example 7.4. Case 1 determines the type of `x` in Listing 7.3. The base type is the generic nominal type `Outer<Int>.Inner<String>` and the type alias `A` is a member of the constrained extension of `Outer.Inner`.

Listing 7.3.: Context substitution map with respect to an extension context

```
struct Outer<T> {
  struct Inner<U> {}
}

extension Outer.Inner where U: Sequence {
  typealias A = (U.Element) -> ()
}

// What is the type of 'x'?
let x: Outer<Int>.Inner<String>.A = ...
```

The generic nominal type `Outer<Int>.Inner<String>` sets `T` to `Int` and `U` to `String`. The extension defines the additional conformance requirement `U: Sequence`. Therefore, the context substitution map with respect to the extension's declaration context is:

Types		
`T`	:=	`Int`
`U`	:=	`String`

Conformances
`String: Sequence`

Applying the above substitution map to the declared interface type of the type alias `A` gives us the final result:

`(U.[Sequence]Element) -> ()`	×	**Types**		
		`T`	:=	`Int`
		`U`	:=	`String`
		Conformances		
		`String: Sequence`		

= `(Character) -> ()`

Example 7.5. In the previous example, we could instead compute the context substitution map for the type declaration context itself. We get almost the same substitution map, except without the conformance requirement:

Types		
`T`	:=	`Int`
`U`	:=	`String`

Applying this substitution map to the declared interface type of the type alias `A` will produce an error type, because the dependent member type `U.[Sequence]Element` is

not a valid type parameter for this substitution map's input generic signature:

`(U.[Sequence]Element) -> ()`	×	**Types**			=	`<<error type>>`
		`T`	:=	`Int`		
		`U`	:=	`String`		

Example 7.6. What if we use the correct declaration context, but the base type does not satisfy the requirements of the constrained extension? For example, consider the type `Outer<Int>.Inner<Int>`. Computing the context substitution map of our base type for the constrained extension's declaration context will output a substitution map containing an invalid conformance, because `Int` does not conform to `Sequence`:

Types		
`T`	:=	`Int`
`U`	:=	`Int`

Conformances
invalid conformance

In fact, the type alias `A` cannot be referenced as a member of this base type at all, because name lookup checks whether the generic requirements of a type declaration are satisfied. Checking generic requirements will be first introduced as part of type resolution (Section 10.1), and will come up elsewhere as well.

Protocol substitution map The context substitution map of a type with respect to a protocol declaration context is called the *protocol substitution map*. Every protocol's generic signature has a single generic parameter with a single conformance requirement, so a substitution map for this generic signature consists of a conformance together with its conforming type. In this manner, there is a one-to-one correspondence between conformances to a specific protocol and the substitution maps of the protocol's generic signature; this mapping is defined by the protocol substitution map construction.

Types		
`Self`	:=	`T`

Conformances
`T: P`

Example 7.7. The type of `x` in Listing 7.4 is determined by the context substitution map of `S` for the protocol declaration context `P`, which is the protocol substitution map for the conformance `S: P`:

Types		
`Self`	:=	`S`

Conformances
`S: P`

Listing 7.4.: The context substitution map with respect to a protocol context

```
struct S: P {
  typealias Element = Int
}

protocol P {
  associatedtype Element
  typealias B = Array<Self.Element>
}

// What is the type of 'x'?
let x: S.B = ...
```

The declared interface type of `B` is `Array<Self.Element>`. Applying our substitution map replaces the dependent member type `Self.Element` with the type witness `Int` from the conformance, giving us the final substituted type `Array<Int>`.

7.2. Composing Substitution Maps

Just as a substitution map can be applied to an original type to produce a substituted type, a substitution map can also be applied to *another substitution map* to produce a new substitution map. The substitution maps are assumed to be *compatible*, meaning the output generic signature of the first must equal the input generic signature of the second. This is called the *composition* of two substitution maps:

$$\boxed{\text{substitution map 1}} \times \boxed{\text{substitution map 2}} = \boxed{\text{substitution map 3}}$$

The action of the composed substitution map is equal to first applying the left hand side substitution map, followed by the right hand side:[1]

$$\boxed{\text{type}} \times \left(\boxed{\text{substitution map 1}} \times \boxed{\text{substitution map 2}}\right) = \left(\boxed{\text{type}} \times \boxed{\text{substitution map 1}}\right) \times \boxed{\text{substitution map 2}}$$

Therefore, the input generic signature of a composed substitution map is the input generic signature of the left hand side; its output generic signature is the output generic

[1] This is why substitution maps act on the right and not the left; it makes our equations more natural.

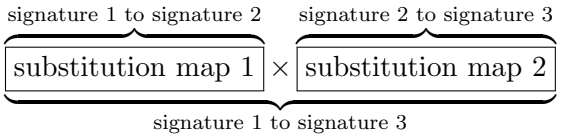
Listing 7.5.: Motivating substitution map composition

```
struct Outer<A> {
  var inner: Inner<Array<A>, A>
}

struct Inner<T, U> {
  var value: (T) -> U
}

let outer: Outer<Int> = ...
let x = outer.inner.value
```

signature of the right hand side.



Composition is defined by applying the second substitution map to each replacement type and conformance of the first substitution map, and collecting the results in a new substitution map. We haven't explained what it means to apply a substitution map to a conformance yet; this will be revisited in Section 8.2.

Example 7.8. Listing 7.5 shows an example where substitution map composition can help reason about the types of chained member reference expressions.

The `inner` stored property of `Outer` has type `Inner<Array<A>, A>`. Here is the context substitution map of this type:

Types		
`T`	:=	`Array<A>`
`U`	:=	`A`

The substitution map's input generic signature is the generic signature of the type declaration `Inner`, which is `<T, U>`.

This type is an interface type for the generic signature of `Outer`, so the output generic signature of the above substitution map is the generic signature of `Outer`, which is `<A>`.

Now, let's look at the `outer` global variable. It has the type `Outer<Int>`, with the following context substitution map:

Types		
`A`	:=	`Int`

The input generic signature of the context substitution map is the generic signature of `Outer`. The output generic signature is the empty generic signature, because the replacement type is fully concrete.

We can compose these two substitution maps, because the first substitution map's output generic signature is the same as the second substitution map's input generic signature. The composition is defined as applying the second substitution map to each replacement type of the first:

Types		
`T`	:=	`Array<A>`
`U`	:=	`A`

$\times$

Types		
`A`	:=	`Int`

$=$

Types		
`T`	:=	`Array<Int>`
`U`	:=	`Int`

Now, the substituted type of `outer.inner.value` is derived from the interface type of `value`, which is the function type `(T) -> U`. Substitution map composition gives us two equivalent ways to compute the substituted type:

1. By applying the first substitution map to the original type `(T) -> U` to get an intermediate substituted type, and then applying the second substitution map to the intermediate substituted type to produce the final substituted type:

$$\left(\texttt{(T) -> U} \times \begin{array}{|l|l|l|} \hline \textbf{Types} \\ \hline \texttt{T} & := & \texttt{Array<A>} \\ \texttt{U} & := & \texttt{A} \\ \hline \end{array} \right) \times \begin{array}{|l|l|l|} \hline \textbf{Types} \\ \hline \texttt{A} & := & \texttt{Int} \\ \hline \end{array}$$

$$= \texttt{Array<A> -> A} \times \begin{array}{|l|l|l|} \hline \textbf{Types} \\ \hline \texttt{A} & := & \texttt{Int} \\ \hline \end{array}$$

$$= \texttt{Array<Int> -> Int}$$

2. By composing the two substitution maps to get a third substitution map, and then applying the third substitution map to the original type `(T) -> U`:

$$\texttt{(T) -> U} \times \left(\begin{array}{|l|l|l|} \hline \textbf{Types} \\ \hline \texttt{T} & := & \texttt{Array<A>} \\ \texttt{U} & := & \texttt{A} \\ \hline \end{array} \times \begin{array}{|l|l|l|} \hline \textbf{Types} \\ \hline \texttt{A} & := & \texttt{Int} \\ \hline \end{array} \right)$$

$$= \texttt{(T) -> U} \times \begin{array}{|l|l|l|} \hline \textbf{Types} \\ \hline \texttt{T} & := & \texttt{Array<Int>} \\ \texttt{U} & := & \texttt{Int} \\ \hline \end{array}$$

$$= \texttt{Array<Int> -> Int}$$

The final substituted type, `Array<Int> -> Int`, is the same in both cases.

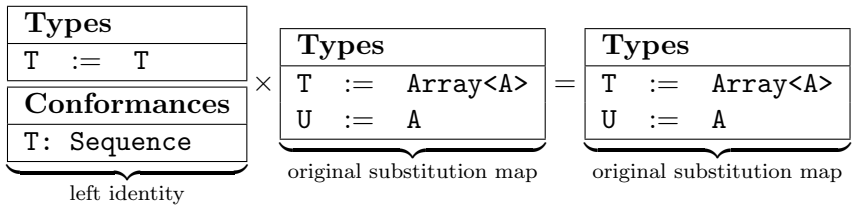
Composing a generic signature's identity substitution map with another substitution map for the same input generic signature leaves the substitution map unchanged:

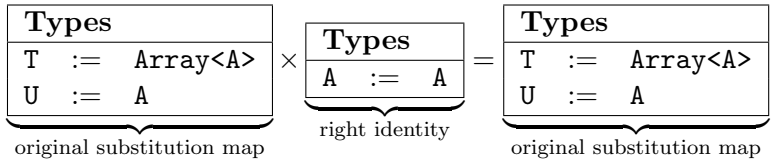
$$\boxed{\text{identity substitution map}} \times \boxed{\text{original substitution map}} = \boxed{\text{original substitution map}}$$

The identity substitution map is also an identity for composition on the right, with the same caveat as for types; it is only true if the other substitution map's replacement types are interface types. If they are contextual types, the archetypes will be replaced with equivalent type parameters.

$$\boxed{\text{original substitution map}} \times \boxed{\text{identity substitution map}} = \boxed{\text{original substitution map}}$$

Example 7.9. The above identities hold for the first substitution map from Example 7.8:





Note that the left and right identity substitution maps are different in this case, because our substitution map has different input and output generic signatures.

Substitution map composition is *associative*. This means that both possible ways of composing three substitution maps will output the same result:

$$\left(\boxed{\text{substitution map 1}} \times \boxed{\text{substitution map 2}}\right) \times \boxed{\text{substitution map 3}}$$
$$= \boxed{\text{substitution map 1}} \times \left(\boxed{\text{substitution map 2}} \times \boxed{\text{substitution map 3}}\right)$$

Mathematical aside These sorts of rules are occasionally useful when writing code in the compiler, but understanding them helps learn to *think* about substitution maps even more. If you have a background in higher math, you will be familiar with the idea of *equational reasoning*; describing a set of objects by writing down the fundamental equations they satisfy.

Linear algebra is the study of vector spaces and linear transformations. A linear transformation is a function from one vector space into another which preserves vector

addition and scalar multiplication. While a vector space over a non-finite field is an infinite set, a linear transformation from a finite-dimensional vector space is completely determined by its values on a finite set of basis vectors.

This is similar in a sense to substitution maps. While the input generic signature of a substitution map might have an infinite set of unique type parameters, the substitution map is not an arbitrary transformation of types; it preserves the "concrete shape" of the original type and transforms dependent member types in a certain way. From this, it follows that the structure of a substitution map is entirely determined by its behavior on a finite set of replacement types and conformances.

An even more mathematical aside In abstract algebra, a *category* is a collection of *objects* and *morphisms* with certain properties. Each morphism is associated with a pair of objects, the *source* and *target*. The set of morphisms with source A and target B is denoted $\mathrm{Hom}(A,\ B)$. The morphisms of a category must obey certain properties:

1. For every object A, there is an *identity morphism* $1_A \in \mathrm{Hom}(A, A)$.
2. If $f \in \mathrm{Hom}(A, B)$ and $g \in \mathrm{Hom}(B, C)$ are a pair of morphisms, there is a third morphism $g \circ f \in \mathrm{Hom}(A, C)$, called the *composition* of f and g.
3. Composition respects the identity: if $f \in \mathrm{Hom}(A, B)$, then $f \circ 1_A = 1_B \circ f = f$.
4. Composition is associative: if $f \in \mathrm{Hom}(A, B)$, $g \in \mathrm{Hom}(B, C)$ and $h \in \mathrm{Hom}(C, D)$, then $h \circ (g \circ f) = (h \circ g) \circ f$.

We can define *the category of vector spaces* by taking the objects to be vector spaces and the morphisms to be linear transformations.

We also can define *the category of generic signatures* where the objects are generic signatures and morphisms are substitution maps, with two caveats. First, the morphism composition notation $(g \circ f)$ is the opposite of our notation for substitution maps $(f \times g)$. Second, in order for the identity substitution map to act as an identity morphism, we need to restrict our category to those substitution maps where the replacement types are interface types only.

We can similarly define a category where the objects are generic signatures and the morphisms are substitution maps containing contextual types only, if we take the identity morphism to be the forwarding substitution map instead of the identity substitution map.

7.3. Building Substitution Maps

Now that you've seen how to get substitution maps from types, and how to compose existing substitution maps, it's time to talk about building substitution maps from scratch using the two variants of the **get substitution map** operation.

The first variant constructs a substitution map directly from its three constituent parts: a generic signature, an array of replacement types, and an array of conformances. The arrays must have the correct length for the given generic signature—equal to the number of generic parameters for the replacement types array, and equal to the number of conformance requirements for the conformances array. The conformances array must satisfy an additional validity condition. Conformances happen to store their conforming type and protocol. Each conformance in a substitution map must match the conformance requirements of the generic signature as follows:

1. The conforming type of a conformance must be canonically equal to the result of applying the substitution map to the subject type of the corresponding conformance requirement.

2. The protocol of a conformance must be the same as the protocol on the right hand side of the corresponding conformance requirement.

This variant of **get substitution map** is used when constructing a substitution map from a deserialized representation, because a serialized substitution map is guaranteed to satisfy the above invariants. It is also used when building a protocol substitution map, because the shape is sufficiently simple—just a single replacement type and a single conformance.

The second variant takes the input generic signature and a pair of callbacks:

1. The **replacement type callback** maps a generic parameter type to a replacement type. It is invoked with each generic parameter type to populate the replacement types array.

2. The **conformance lookup callback** maps a protocol conformance requirement to a conformance. It is invoked with each conformance requirement to populate the conformances array.

The conformance lookup callback takes three parameters:

1. The *original type*; this is the subject type of the conformance requirement.

2. The *substituted type*; this is the result of applying the substitution map to the original type, which should be canonically equal to the conforming type of the conformance that will be returned.

3. The protocol declaration named by the conformance requirement.

The callbacks can be arbitrarily defined by the caller. Several pre-existing "functors" also implement common behaviors. For the replacement type callback,

1. The **query substitution map** functor looks up a generic parameter in an existing substitution map.

2. The **query type map** functor looks up a generic parameter in an LLVM `DenseMap`.

For the conformance lookup callback,

1. The **global conformance lookup** functor performs a global conformance lookup (Section 8.1).

2. The **local conformance lookup** functor performs a local conformance lookup into another substitution map (Section 8.4).

3. The **make abstract conformance** functor asserts that the substituted type is a type variable, type parameter or archetype, and returns an abstract conformance (also in Section 8.4). It is used when it is known that the substitution map can be constructed without performing any conformance lookups, as is the case with the identity substitution map.

Specialized types only store their generic arguments, not conformances, so the context substitution map of a specialized type is constructed by first populating a `DenseMap` with the generic arguments of the specialized type and all of its parent types, and then invoking the **get substitution map** operation with the **query type map** and **global conformance lookup** functors.

The identity substitution map of a generic signature is constructed from a replacement type callback which just returns the input generic parameter together with the **make abstract conformance** functor.

Example 7.10. A substitution map which does *not* satisfy the invariants specified above, and thus cannot be constructed. First, the generic signature:

```
<T where T: Sequence, T.[Sequence]Element: Comparable>:
```

And the substitution map:

Types
`T := Array<Int>`

Conformances
`Array<Int>: Sequence`
`String: Comparable`

The generic signature has two conformance requirements:

`T: Sequence`

`T.[Sequence]Element: Comparable`

Applying the substitution map to the subject type of each requirement produces the expected conforming types:

`T` × substitution map = `Array<Int>`

`T.[Sequence]Element` × substitution map = `Int`

The actual conforming type of the second conformance violates our invariant:

Actual	Expected	Correct?
`Array<Int>`	`Array<Int>`	✓
`String`	`Int`	×

7.4. Nested Nominal Types

Nominal type declarations can appear inside other declaration contexts, subject to the following restrictions:

1. Structs, enums and classes cannot be nested in generic local contexts.
2. Structs, enums and classes cannot be nested in protocols or protocol extensions.
3. Protocols cannot be nested in any declaration context other than a source file.

We're going to explore the implementation limitations behind these restrictions, and possible future directions for lifting them. (The rest of the book talks about what the compiler does, but this section is about what the compiler *doesn't* do.)

Types in generic local contexts This restriction is a consequence of a shortcoming in the representation of a nominal type. Recall from Chapter 3 that nominal types and generic nominal types store a parent type, and generic nominal types additionally store a list of generic arguments, corresponding to the generic parameter list of the nominal type declaration. This essentially means there is no place to store the generic arguments from outer generic local contexts.

Listing 7.6 shows a nominal type nested inside of a generic function. The generic signature of `Nested` contains the generic parameter `T` from the outer generic function `algorithm()`. However, under our rules, the declared interface type of `Nested` is a singleton nominal type, because `Nested` does not have its own generic parameter list, and its parent context is not a nominal type declaration. This means there is no way to recover a context substitution map for this type because the generic argument for `T` is not actually stored anywhere.

In the source language, there is no way to specialize `Nested`; the reference to `T` inside `f()` is always understood to be the generic parameter `T` of the outer function. However,

Listing 7.6.: A nominal type declaration in a generic local context

```
func f<T>(t: T) {
  struct Nested {
    let t: T

    func printT() {
      print(t)
    }
  }

  Nested(t: t).printT()
}

func g() {
  f(t: 123)
  f(t: "hello")
}
```

inside the compiler, different generic specializations can still arise. If the two calls to `f()` from inside `g()` are specialized and inlined by the SIL optimizer for example, the two temporary instances of `Nested` must have different in-memory layouts, because in one call `T` is `Int`, and in the other `T` is `String`.

A better representation for the specializations of nominal types would replace the parent type and list of generic arguments with a single "flat" list that includes all outer generic arguments as well. This approach could represent generic arguments coming from outer local contexts without loss of information.

Luckily, this "flat" representation is already implemented in the Swift runtime. The runtime type metadata for a nominal type includes all the generic parameters from the nominal type declaration's generic signature, not just the generic parameters of the nominal type declaration itself. So while lifting this restriction would require some engineering effort on the compiler side, it would be a backward-deployable and ABI-compatible change.

Types in protocol contexts To allow struct, enum and class declarations to appear inside protocols and protocol extensions, a decision needs to be made as to whether the protocol `Self` type should be "captured" by the nested type.

If the nested type captures `Self`, the code shown in Listing 7.6 would become valid. With this model, the `Nested` struct depends on `Self`, so it would not make sense to

Listing 7.7.: A nominal type declaration nested in a protocol context

```
protocol P {}

extension P {
  struct Nested {
    let value: Self

    func method() {
      print(value)
    }
  }

  func f() {
    Nested(value: self).method()
  }
}

struct S: P {}
```

reference it as a member of the protocol itself, like `P.Nested`. Instead, `Nested` would behave as if it was a member of every conforming type, like `S.Nested` above (or even `T.Nested`, if `T` is a generic parameter conforming to `P`).

At the implementation level, this would mean that the generic signature of a nominal type nested inside of a protocol context would include the protocol `Self` type, and the *entire* parent type, for example `S` in `S.Nested`, would become the replacement type for `Self` in the context substitution map.

The alternative approach would prohibit the nested type from referencing the protocol `Self` type. The nested type's generic signature would *not* include the protocol `Self` type, and `P.Nested` would be a valid member type reference. The protocol would effectively act as a namespace for the nominal types it contains, with the nested type not depending on the conformance to the protocol in any way.

Protocols in other declaration contexts The final generalization is the ability to nest protocols inside other declaration contexts, such as functions or nominal types. This can be broken down into two cases:

1. Protocols inside non-generic declaration contexts.

2. Protocols inside generic declaration contexts.

Listing 7.8.: Protocol declaration nested inside other declaration contexts

```
struct Outer {
  protocol P {
    func f()
  }
}

func generic<T>(_: T) {
  protocol P {
    // What does this mean?
    func f(_: T)
  }
}
```

Listing 7.8 shows both possibilities. The first case is a relatively straightforward; the non-generic declaration contexts acts as a namespace to which the protocol declaration is scoped.

In contrast, the second case would introduce significant complexity to the language design, by allowing "generic protocols" with more generic parameters than just the protocol `Self` type. Such a protocol would be what Haskell calls a "multi-parameter type class." Unlike the prior generalizations this one carries profound implications and tradeoffs and it is not clear that it belongs in the Swift language.

7.5. Source Code Reference

Key source files:

- include/swift/AST/SubstitutionMap.h
- lib/AST/SubstitutionMap.cpp

Other source files:

- include/swift/AST/GenericSignature.h
- include/swift/AST/Type.h
- include/swift/AST/Types.h

Type	*class*

See also Section 3.6.

- `subst()` applies a substitution map to this type and returns the substituted type.

`TypeBase` *class*

See also Section 3.6 and Section 6.5.

- `getContextSubstitutionMap()` returns this type's context substitution map with respect to the given `DeclContext`.

`SubstitutionMap` *class*

Represents an immutable, uniqued substitution map.

As with `Type` and `GenericSignature`, this class stores a single pointer, so substitution maps are cheap to pass around as values. The default constructor `SubstitutionMap()` constructs an empty substitution map. The implicit `bool` conversion tests for a non-empty substitution map.

Accessor methods:

- `empty()` answers if this is the empty substitution map; this is the logical negation of the `bool` implicit conversion.
- `getGenericSignature()` returns the substitution map's input generic signature.
- `getReplacementTypes()` returns an array of `Type`.
- `hasAnySubstitutableParams()` answers if the input generic signature contains at least one generic parameter not fixed to a concrete type; that is, it must be non-empty and not fully concrete (see the `areAllParamsConcrete()` method of `GenericSignatureImpl` from Section 6.5).

Recursive properties computed from replacement types:

- `hasArchetypes()` answers if any of the replacement types contain a primary archetype or opened existential archetype.
- `hasOpenedExistential()` answers if any of the replacement types contain an opened existential archetype.
- `hasDynamicSelf()` answers if any of the replacement types contain the dynamic Self type.

Canonical substitution maps:

- `isCanonical()` answers if the replacement types and conformances stored in this substitution map are canonical.

- `getCanonical()` constructs a new substitution map by canonicalizing the replacement types and conformances of this substitution map.

Composing substitution maps (Section 7.2):

- `subst()` applies another substitution map to this substitution map, producing a new substitution map.

Two overloads of the `get()` static method are defined for constructing substitution maps (Section 7.3).

`get(GenericSignature, ArrayRef<Type>, ArrayRef<ProtocolConformanceRef>)` builds a new substitution map from an input generic signature, an array of replacement types, and array of conformances.

`get(GenericSignature, TypeSubstitutionFn, LookupConformanceFn)` builds a new substitution map by invoking a pair of callbacks to produce each replacement type and conformance.

`TypeSubstitutionFn` *type alias*

The type signature of a replacement type callback for `SubstitutionMap::get()`.

```
using TypeSubstitutionFn
  = llvm::function_ref<Type(SubstitutableType *dependentType)>;
```

The parameter type is always a `GenericTypeParamType *` when the callback is used with `SubstitutionMap::get()`.

`QuerySubstitutionMap` *struct*

A functor intended to be used with `SubstitutionMap::get()` as a replacement type callback. Overloads `operator()` with the signature of `TypeSubstitutionFn`.

Constructed from a `SubstitutionMap`:

```
1 QuerySubstitutionMap{subMap}
```

`QueryTypeSubstitutionMap` *struct*

A functor intended to be used with `SubstitutionMap::get()` as a replacement type callback. Overloads `operator()` with the signature of `TypeSubstitutionFn`.

Constructed from an LLVM `DenseMap`:

```
1 DenseMap<SubstitutableType *, Type> typeMap;
2
3 QueryTypeSubstitutionMap{typeMap}
```

`LookupConformanceFn` *type alias*

The type signature of a conformance lookup callback for `SubstitutionMap::get()`.

```
using LookupConformanceFn = llvm::function_ref<
    ProtocolConformanceRef(CanType origType,
                           Type substType,
                           ProtocolDecl *conformedProtocol)>;
```

`LookUpConformanceInModule` *struct*

A functor intended to be used with `SubstitutionMap::get()` as a conformance lookup callback. Overloads `operator()` with the signature of `LookupConformanceFn`.

Constructed with a `ModuleDecl *`:

```
1 LookUpConformanceInModule{moduleDecl}
```

`LookUpConformanceInSubstitutionMap` *struct*

A functor intended to be used with `SubstitutionMap::get()` as a conformance lookup callback. Overloads `operator()` with the signature of `LookupConformanceFn`.

Constructed with a `SubstitutionMap`:

```
1 LookUpConformanceInSubstitutionMap{subMap}
```

`MakeAbstractConformance` *struct*

A functor intended to be used with `SubstitutionMap::get()` as a conformance lookup callback. Overloads `operator()` with the signature of `LookupConformanceFn`.

Constructed without arguments:

```
1 MakeAbstractConformance()
```

`GenericSignature` *class*

See also Section 6.5.

- `getIdentitySubstitutionMap()` returns the substitution map that replaces each generic parameter with itself.

8. Conformances

A *conformance* describes how a type satisfies the requirements of a protocol. In the previous chapter, you saw that conformances appear in substitution maps, populated by a global conformance lookup operation. Now, we will discuss their structure and the role that conformances play in type substitution, and look at global conformance lookup in more detail. There are three kinds of conformance:

1. An **invalid conformance** denotes that a type does not actually conform to the protocol.
2. An **abstract conformance** denotes that a type conforms to the protocol, but it is not known where this conformance was declared. Described in Section 8.4.
3. A **concrete conformance** represents a conformance with a known definition.

Concrete conformances are further broken down into four sub-kinds, with the first two sub-kinds being the main focus of this chapter:

1. A **normal conformance** represents the actual declaration of a conformance on a type or extension.
2. A **specialized conformance** is how an arbitrary specialized type conforms to a protocol.
3. A **self conformance** is how a protocol conforms to itself, which is only possible in a few very special cases. Described in Section 15.2.
4. An **inherited conformance** is how a subclass conforms to a protocol when the conformance was declared on the superclass. Described in Section 16.1.

Normal conformances Structs, enums and classes can conform to protocols. A normal conformance represents the *declaration* of such a conformance. Normal conformances are declared by referencing a protocol from the inheritance clause of a type or extension declaration:

```
1 struct Horse: Animal {...}
2
3 struct Cow {...}
4 extension Cow: Animal {...}
```

Each type or extension declaration has a list of *local conformances*, which are the normal conformances declared on that type or extension. In the above, the struct declaration `Horse` has a single local conformance. The struct declaration `Cow` does not have any local conformances itself, but the *extension* of `Cow` has one.

Nominal type declarations have a *conformance lookup table*, which stores the local conformances of the type and any of its extensions, together with conformances inherited from the superclass, if the type declaration is a class declaration. Extension declarations do not have a conformance lookup table of their own; their local conformances are part of the extended type's conformance lookup table. The conformance lookup table is used to implement global conformance lookup. The rest of the compiler does not interact directly with conformance lookup tables.

Broken down into constituent parts, a normal conformance stores the following:

- **The type:** the declared interface type of the conforming context.
- **The protocol:** this is the protocol being conformed to.
- **The conforming context:** either a nominal type declaration (if the conformance is stated on the type) or an extension thereof (if the conformance is stated on an extension).
- **The generic signature:** the generic signature of the conforming context. If the conformance context is a nominal type declaration or an unconstrained extension, this is the generic signature of the nominal type. If the conformance context is a constrained extension, this generic signature will have additional requirements, and the conformance becomes a conditional conformance. Conditional conformances are described in Section 12.2.
- **Type witnesses:** a mapping from each associated type of the protocol to the concrete type witnessing the associated type requirement. This is an interface type written in terms of the generic signature of the conformance. Section 8.3 will talk about type witnesses.
- **Associated conformances:** a mapping from the conformance requirements of the requirement signature to a conformance of the substituted subject type to the requirement's protocol. Section 8.5 will talk about associated conformances.
- **Value witnesses:** for each value requirement of the protocol, the declaration witnessing the requirement. This declaration is either a member of the conforming nominal type, an extension of the conforming nominal type, or it is a default implementation from a protocol extension. The mapping is more elaborate than just storing the witness declaration; Chapter 17 goes into the details.

Specialized conformances In Section 7.2, you saw that substitution maps apply to types, other substitution maps, and conformances. Applying a non-identity substitution map to a normal conformance produces a *specialized conformance*, which wraps the underlying normal conformance together with the substitution map, which we call the *conformance substitution map*:

$$\boxed{\text{normal conformance}} \times \boxed{\text{substitution map}} = \boxed{\text{specialized conformance}}$$

The conformance substitution map is never the identity substitution map; applying the identity substitution map to a normal conformance simply returns the original normal conformance. This avoids the overhead of constructing a specialized conformance in this case, and also has a nice mathematical interpretation: it ensures that the identity substitution map actually acts as the identity on a normal conformance:

$$\boxed{\text{normal conformance}} \times \boxed{\text{identity substitution map}} = \boxed{\text{normal conformance}}$$

Canonical conformances Like types and substitution maps, specialized conformances are immutable and uniqued. A specialized conformance is *canonical* if the substitution map is canonical. Canonicalizing a specialized conformance returns a new conformance with the same underlying conformance, and the canonicalized conformance substitution map. Normal conformances are always canonical.

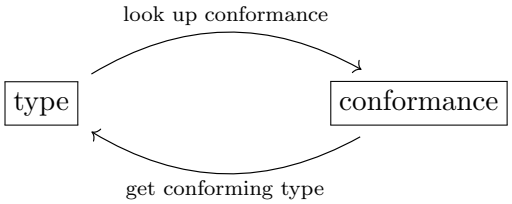
8.1. Conformance Lookup

Conformances are typically found via *global conformance lookup*, which takes a type and a protocol and returns a conformance. Global conformance lookup answers the question "does a type conform to a protocol"—when the result is an invalid conformance, the type does not conform, otherwise it conforms.

Global conformance lookup can be understood as the action of a protocol declaration on the left of a type:

$$\boxed{\text{protocol declaration}} \times \boxed{\text{type}} = \boxed{\text{conformance}}$$

All conformances store the conformed protocol and conforming type. The conforming type of a valid conformance found by global conformance lookup is canonical-equal to the type that was handed to the lookup operation (the two types might differ by type sugar, so are not required to be pointer-equal). We can exhibit this with a *commutative diagram*:



A commutative diagram is a diagram where every path with the same start and end leads to the same result. The above commutative diagram shows two pairs of paths:

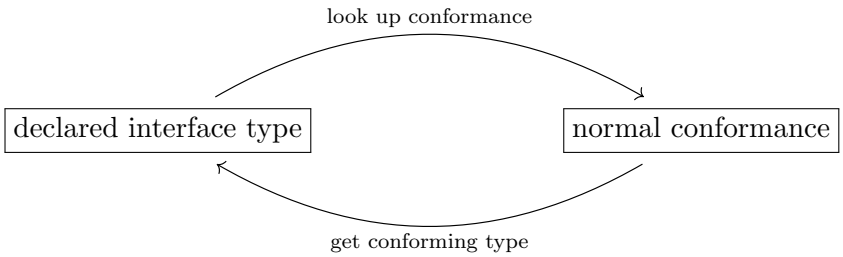
1. Starting from a type, we look up its conformance to a fixed protocol, and get the conforming type of this conformance. This takes us back to the original type.

2. Starting from a conformance, we get its conforming type, and perform a global conformance lookup with this type and our protocol. This gives us the original conformance.

Global conformance lookup always returns a normal conformance when given the declared interface type of a type declaration that directly conforms to the protocol:

$$\boxed{\text{protocol declaration}} \times \boxed{\text{declared interface type}} = \boxed{\text{normal conformance}}$$

The conforming type of a normal conformance is the declared interface type; plugging this information into our commutative diagram gives us the following:



You will see more equations and commutative diagrams in the next section, after a brief interlude where we discuss a conceptual difficulty.

Coherence In reality, our diagram above hand-waves away a significant complication. Since a conformance can be declared on an extension, and the extended type might be defined in a different module, it is possible that two modules may define the same conformance in two different ways. Global conformance lookup is not guaranteed to be *coherent*.

For example, imagine if there were two different conformances of some concrete type `K` to `Hashable`. Then it would be possible for two different modules to construct values of type `Set<K>` with incompatible hash functions; passing such a value from one module to the other would result in undefined behavior.

For now, there's no real answer to this dilemma. The compiler rejects duplicate conformance definitions if an existing conformance is statically visible, so this scenario cannot occur with `Int` and `Hashable` for instance, because the conformance of `Int` to `Hashable` in the standard library is always visible, so any attempt to define a new conformance would be diagnosed as an error.

However, if the concrete type `K` is defined in some common module, and two separately-compiled modules both define a conformance of `K` to `Hashable`, a module that imports all three will observe both conformances statically, with unpredictable results.

A similar scenario can occur with library evolution. Suppose a library publishes the concrete type `K`, and a third party defines a conformance of `K` to `Hashable`. If the library vendor then adds their own conformance of `K` to `Hashable`, the previously-compiled client might encounter incorrect behavior at runtime.

The global conformance lookup operation as implemented by the compiler actually takes a module declaration as an input, along with the type and protocol. The intent behind passing the module was that it should be taken into account somehow, perhaps restricting the search to those conformances that are transitively visible via import declarations, with an error diagnostic in the case of a true ambiguity. At the time of writing, this module declaration is ignored.

The runtime equivalent of a global conformance lookup is a *dynamic cast* from a concrete type to an existential type. Dynamic casts suffer from a similar ambiguity issue. To be coherent, this dynamic cast operation would need to inspect something akin to a module import graph reified at the call site to be able to disambiguate duplicate conformances.

In the absence of proper compiler support for addressing this problem, there is a rule of thumb that, if followed by Swift users, mostly guarantees coherence. The rule is that when defining a conformance on an extension, either the extended type or the protocol should be part of the current module.

That is, the following is fine, because our own type conforms to a standard library protocol:

```
1 struct MyType {...}
2 extension MyType: Hashable {...}
```

This is fine too, because a standard library type conforms to our own protocol:

```
1 protocol MyProtocol {...}
2 extension Int: MyProtocol {...}
```

However the next example is potentially problematic; we're defining the conformance of a standard library type to a standard library protocol, and nothing prevents some other module from declaring the same conformance:

```
1 extension String.UTF8View: Hashable {...}
```

A conformance where neither the conforming type nor the protocol is part of the current module is called a *retroactive conformance*. Today, retroactive conformances are

allowed without any restrictions. In a future compiler version, they might generate a warning.

Unfortunately, avoiding retroactive conformances does not completely solve the issue either, because there is another possible hole with class inheritance and library evolution. Consider a framework which defines an open class and a protocol:

```
1 public protocol MyProtocol {}
2 open class BaseClass {}
```

A client might declare a subclass of `BaseClass` and conform it to `MyProtocol`, concluding this it is safe to do so because the conforming type, `DerivedClass`, is owned by the client, and thus this is not a retroactive conformance:

```
1 import OtherLibrary
2
3 class DerivedClass: BaseClass {}
4 extension DerivedClass: MyProtocol {}
```

However, in the next version of the framework, the framework author might decide to conform `BaseClass` to `MyProtocol`. At this point, `DerivedClass` has two duplicate conformances to `MyProtocol`; the inherited conformance from `BaseClass`, and the local conformance of `DerivedClass`.

8.2. Conformance Substitution

If global conformance lookup returns a normal conformance when given the declared interface type of a nominal type declaration, the natural question is what it should return given an arbitrary specialized type of a nominal type declaration. As you might guess, the answer is that it returns a specialized conformance.

Recall the following three equations:

1. First, the factorization of a specialized type into the declared interface type of some nominal type declaration, together with a substitution map, from Section 7.1:

$$\boxed{\text{declared interface type}} \times \boxed{\text{substitution map}} = \boxed{\text{specialized type}}$$

2. Next, the notation for global conformance lookup from the previous section:

$$\boxed{\text{protocol declaration}} \times \boxed{\text{type}} = \boxed{\text{conformance}}$$

3. And finally, the fact that global conformance lookup returns a normal conformance when given a declared interface type, also from the previous section:

$$\boxed{\text{protocol declaration}} \times \boxed{\text{declared interface type}} = \boxed{\text{normal conformance}}$$

We want to answer the question of what it means to perform a global conformance lookup with an arbitrary specialized type:

$$\boxed{\text{protocol declaration}} \times \boxed{\text{specialized type}} = \boxed{?}$$

If we substitute equation (1) into the above, we get the following:

$$\begin{aligned}&\boxed{\text{protocol declaration}} \times \boxed{\text{specialized type}} \\ &= \boxed{\text{protocol declaration}} \times \Big(\boxed{\text{declared interface type}} \times \boxed{\text{substitution map}} \Big)\end{aligned}$$

Now, here's the trick. A binary operation $\times$ is *associative* if the placement of parentheses doesn't matter; that is, if $(A \times B) \times C = A \times (B \times C)$. We want global conformance lookup and type substitution to be associative. This means we should be able to change the placement of the parentheses in the above equation while getting the same result:

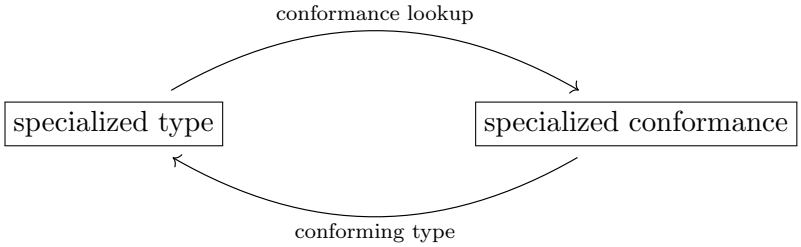
$$\begin{aligned}&\boxed{\text{protocol declaration}} \times \boxed{\text{specialized type}} \\ &= \boxed{\text{protocol declaration}} \times \Big(\boxed{\text{declared interface type}} \times \boxed{\text{substitution map}} \Big) \\ &= \Big(\boxed{\text{protocol declaration}} \times \boxed{\text{declared interface type}} \Big) \times \boxed{\text{substitution map}}\end{aligned}$$

Now, by equation (3), the term inside the parentheses gives us a normal conformance:

$$\begin{aligned}&\boxed{\text{protocol declaration}} \times \boxed{\text{specialized type}} \\ &= \boxed{\text{protocol declaration}} \times \Big(\boxed{\text{declared interface type}} \times \boxed{\text{substitution map}} \Big) \\ &= \Big(\boxed{\text{protocol declaration}} \times \boxed{\text{declared interface type}} \Big) \times \boxed{\text{substitution map}} \\ &= \boxed{\text{normal conformance}} \times \boxed{\text{substitution map}}\end{aligned}$$

So there you go—global conformance lookup with a specialized type returns a specialized conformance, whose conformance substitution map is the context substitution map of the specialized type.

Similarly, the conforming type of a specialized conformance is the specialized type we get by applying the conformance substitution map to the conforming type of the normal conformance; this has to be the case, because of our commutative diagram:



You've now seen what it means to apply a substitution map to a normal conformance, and how this operation arises naturally in the implementation of global conformance lookup. As it turns out, you can apply substitution maps to specialized conformances as well.

$$\boxed{\text{specialized conformance}} \times \boxed{\text{substitution map 2}} = \boxed{?}$$

(There are going to be two substitution maps in play now, the conformance substitution map of the specialized conformance, and the substitution map being applied, so let's label them "substitution map 1" and "substitution map 2.")

First, we need a notion of the output generic signature of a conformance; we require that the input generic signature of the applied substitution map is the output generic signature of the conformance. The output generic signature of a normal conformance is the generic signature of the conformance context. The output generic signature of a specialized conformance is the output generic signature of its conformance substitution map. As with substitution maps and interface types, the output generic signature of a specialized conformance isn't actually stored; it is implicit from usage.

With this out of the way, we can proceed to derive our equation. First, we factor the specialized conformance into a normal conformance together with a substitution map:

$$\begin{aligned}&\boxed{\text{specialized conformance}} \times \boxed{\text{substitution map 2}} \\ &= \Big(\boxed{\text{normal conformance}} \times \boxed{\text{substitution map 1}}\Big) \times \boxed{\text{substitution map 2}}\end{aligned}$$

Once again, we just move the parentheses around, because it intuitively appears to make sense:

$$\begin{aligned}&\boxed{\text{specialized conformance}} \times \boxed{\text{substitution map 2}} \\ &= \Big(\boxed{\text{normal conformance}} \times \boxed{\text{substitution map 1}}\Big) \times \boxed{\text{substitution map 2}} \\ &\quad = \boxed{\text{normal conformance}} \times \Big(\boxed{\text{substitution map 1}} \times \boxed{\text{substitution map 2}}\Big)\end{aligned}$$

Finally, we can simplify inside of the parentheses on the third line above, by composing the two substitution maps. This gives us our answer: applying a substitution map to a specialized conformance builds a new specialized conformance with the same underlying normal conformance, and a new conformance substitution map obtained by composing the old conformance substitution map with the given substitution map.

Recall that substitution map composition is associative:

$$\begin{aligned}&\Big(\boxed{\text{substitution map 1}} \times \boxed{\text{substitution map 2}}\Big) \times \boxed{\text{substitution map 3}} \\ &\quad = \boxed{\text{substitution map 1}} \times \Big(\boxed{\text{substitution map 2}} \times \boxed{\text{substitution map 3}}\Big)\end{aligned}$$

The above together with everything else from this section can be combined into one final identity. You just saw the same identity for normal conformances; it holds for specialized conformances too:

$$\Big(\boxed{\text{specialized conformance}} \times \boxed{\text{substitution map 2}}\Big) \times \boxed{\text{substitution map 3}}$$
$$= \boxed{\text{specialized conformance}} \times \Big(\boxed{\text{substitution map 2}} \times \boxed{\text{substitution map 3}}\Big)$$

8.3. Type Witnesses

A concrete type fulfills the associated type requirements of a protocol by declaring a *type witness* for each associated type. Type witnesses are declared in one of four ways:

1. Via a **member type declaration** having the same name as the associated type. Usually this member type is a type alias, but it is legal to use a nested nominal type declaration as well. If the conforming type is a class, the member type may also be defined in a superclass.

2. Via a **generic parameter** having the same name as the associated type. If the conforming type is not generic but is nested inside of a generic context, a generic parameter of the innermost generic context can be used.[1]

3. Via **associated type inference**, where it is implicitly derived from the declaration of a witness to a value requirement.

4. Via a **default type witness** on the associated type declaration, which is used if all else fails.

The conformance checker is responsible for resolving type witnesses and ensuring they satisfy the requirements of the protocol's requirement signature, as described earlier in Section 6.1. The problem of checking whether concrete types satisfy generic requirements is covered in Section 10.2.

Example 8.1. Listing 8.1 illustrates all four possibilities. In all cases other than the first, the conformance checker synthesizes a type alias declaration with the same name as the associated type. This type alias declaration is visible as a member of the concrete conforming type. For this reason, it appears at first glance that the generic parameter `T` is a member type of `WithGenericParam<T>`:

```
1 func squared(_ x: WithGenericParam<Int>.T) -> Int {
2   return x * x
3 }
```

[1] The latter being allowed was probably an oversight, but it's the behavior implemented today.

Listing 8.1.: Different ways of declaring a type witness in a conformance

```
protocol P {
  associatedtype T = Int
  func f(_: T)
}

extension P {
  func f(_: T) {}
}

struct WithMemberType: P {
  struct T {}
}

struct WithGenericParam<T>: P {}

struct WithInferredType: P {
  func f(_: String) {}
}

struct WithDefault: P {}
```

However, the member type `T` is not the generic parameter declaration itself, but the synthesized type alias declaration. If `WithGenericParam` did not declare a conformance to `P`, there would be no member type named `T`, because generic parameter declarations are not visible as member types.

Projection Given a conformance and an associated type of the conformed protocol, we can ask the conformance for the corresponding type witness. The next section will explain how type substitution of dependent member types uses the type witnesses of a conformance, but first we need to develop the "algebra" of type witnesses.

We can understand getting a type witness out of a conformance as the action of an associated type declaration (of a protocol) on the left of a conformance (to this protocol):

$$\boxed{\text{associated type}} \times \boxed{\text{conformance}} = \boxed{\text{type witness}}$$

Normal conformances directly store type witnesses as interface types for the conforming context's generic signature; getting a type witness from a normal conformance projects this stored value:

$$\boxed{\text{associated type}} \times \boxed{\text{normal conformance}} = \boxed{\text{type witness}}$$

Next, we need to understand what it means to get a type witness from a specialized conformance:

$$\boxed{\text{associated type}} \times \boxed{\text{specialized conformance}} = \boxed{?}$$

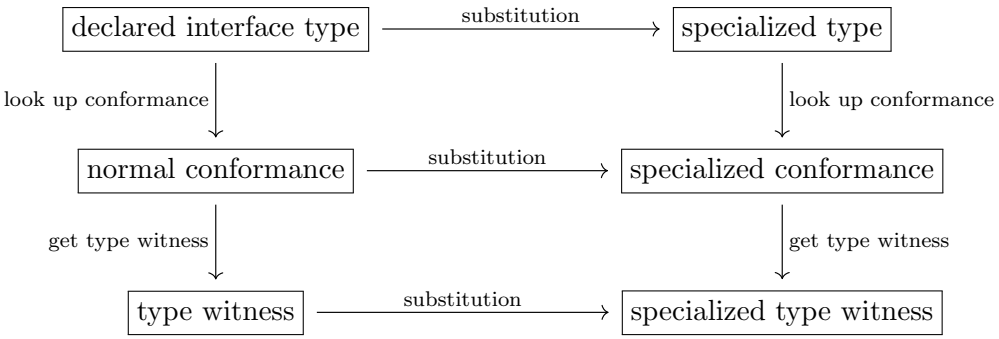
As it turns out, this is implemented by applying the conformance substitution map to the corresponding type witness from the underlying normal conformance. To understand why, we start by writing the specialized conformance as a substitution map applied to a normal conformance:

$$\begin{aligned}&\boxed{\text{associated type}} \times \boxed{\text{specialized conformance}} \\ &= \boxed{\text{associated type}} \times \Big(\boxed{\text{normal conformance}} \times \boxed{\text{substitution map}} \Big)\end{aligned}$$

Then, we repeat our magic trick—we want this action to be associative, so we move the parentheses around:

$$\begin{aligned}&\boxed{\text{associated type}} \times \boxed{\text{specialized conformance}} \\ &= \boxed{\text{associated type}} \times \Big(\boxed{\text{normal conformance}} \times \boxed{\text{substitution map}} \Big) \\ &= \Big(\boxed{\text{associated type}} \times \boxed{\text{normal conformance}} \Big) \times \boxed{\text{substitution map}}\end{aligned}$$

Figure 8.1.: Type witnesses of normal and specialized conformances



Therefore, the problem of projecting a type witness of a specialized conformance reduces to applying the conformance substitution map to a type witness of the underlying normal conformance:

$$\begin{aligned}
&\boxed{\text{associated type}} \times \boxed{\text{specialized conformance}} \\
&= \boxed{\text{associated type}} \times \Big(\boxed{\text{normal conformance}} \times \boxed{\text{substitution map}} \Big) \\
&= \Big(\boxed{\text{associated type}} \times \boxed{\text{normal conformance}} \Big) \times \boxed{\text{substitution map}} \\
&= \boxed{\text{type witness}} \times \boxed{\text{substitution map}}
\end{aligned}$$

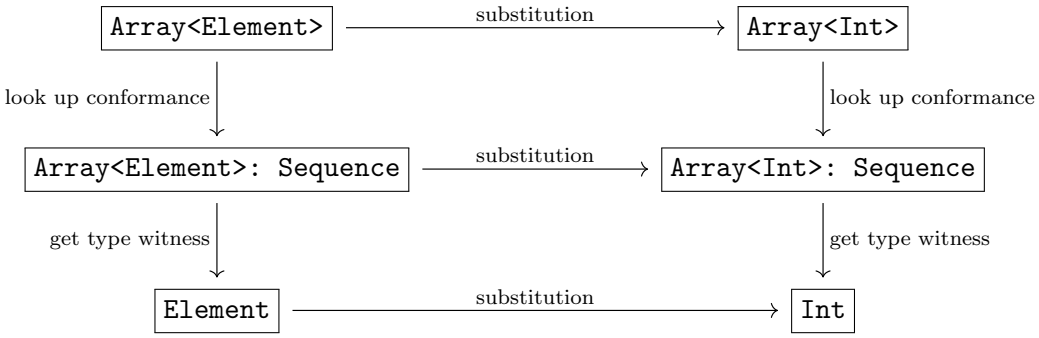
The above equations show that getting a type witness of a specialized conformance fits nicely with our notational formalism. Another way to convince yourself that this makes sense is with a commutative diagram. Figure 8.1 shows a commutative diagram relating global conformance lookup with getting a type witness from a specialized conformance:

1. Starting from the declared interface type of a nominal type declaration, we can look up the conformance to a protocol, and get the type witness for an associated type out of this conformance.

2. If we apply a substitution map to the declared interface type, we get a specialized type. Global conformance lookup with the specialized type returns a specialized conformance. Getting a type witness from the specialized conformance applies the substitution map to the type witness of the normal conformance.

3. Each horizontal arrow applies *the same* substitution map, which is the context substitution map of the specialized type.

Figure 8.2.: Type witnesses of the conformances of `Array<Element>` and `Array<Int>` to `Sequence`



We saw that the type witnesses of a normal conformance are written in terms of the conforming context's generic signature. For a specialized conformance, they are written in terms of the output generic signature of the conformance substitution map.

Example 8.2. To make this concrete, say we look up the conformance of `Array<Int>` to `Sequence`, and then get the type witness for the `Element` associated type. The declared interface type of `Array` is `Array<Element>`, where `Element` is the generic parameter of `Array`. The type witness of the `Element` associated type in the normal conformance of `Array` to `Sequence` is the `Element` generic parameter type.

Our specialized type is `Array<Int>`. The context substitution map of `Array<Int>` replaces `Element` with `Int`:

Types		
`Element`	:=	`Int`

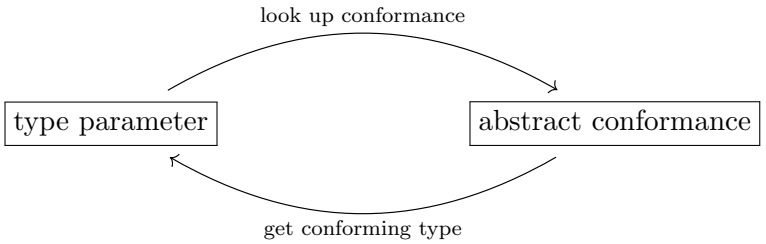
Figure 8.2 shows the commutative diagram for this case. Each horizontal arrow in the commutative diagram represents the application of this substitution map to a type or conformance. Since the diagram is commutative, we can start at the top left and always end up at the bottom right, independent of which of the three paths we take.

8.4. Abstract Conformances

An *abstract conformance* represents the conformance of a type parameter (or archetype) to a protocol, where the type parameter is understood to satisfy `requiresProtocol()` generic signature query of some generic signature:

protocol declaration × type parameter = abstract conformance

The conforming type of an abstract conformance is a type parameter. Similar to normal conformances and specialized conformances, we can show the relationship between an abstract conformance and its conforming type with a commutative diagram:



Abstract conformances allow us to formalize the behavior of type substitution with a dependent member type:

$$\boxed{\text{dependent member type}} \times \boxed{\text{substitution map}} = \boxed{?}$$

Before we can solve the above, consider a bound dependent member type `T.[P]A` in some generic signature, with base type `T` and associated type `A` of protocol `P`. If the compiler was able to form this dependent member type, either by type resolution or some other means, it necessarily follows that `T` conforms to `P` in the type parameter's generic signature. This allows us to factor the dependent member type into an associated type declaration together with an abstract conformance:

$$\boxed{\texttt{[P]A}} \times \boxed{\texttt{T: P}} = \boxed{\texttt{T.[P]A}}$$

In order words, dependent member types are the type witnesses of abstract conformances:

$$\boxed{\text{associated type}} \times \boxed{\text{abstract conformance}} = \boxed{\text{dependent member type}}$$

This gives us the following equation:

$$\begin{aligned}&\boxed{\text{dependent member type}} \times \boxed{\text{substitution map}} \\ &= \left(\boxed{\text{associated type}} \times \boxed{\text{abstract conformance}}\right) \times \boxed{\text{substitution map}}\end{aligned}$$

Next we switch the parentheses around:

$$\begin{aligned}&\boxed{\text{dependent member type}} \times \boxed{\text{substitution map}} \\ &= \left(\boxed{\text{associated type}} \times \boxed{\text{abstract conformance}}\right) \times \boxed{\text{substitution map}} \\ &\qquad = \boxed{\text{associated type}} \times \left(\boxed{\text{abstract conformance}} \times \boxed{\text{substitution map}}\right)\end{aligned}$$

You saw how normal and specialized conformances are substituted in Section 8.2. Now, it appears we need the ability to apply a substitution map to an abstract conformance. This operation is called *local conformance lookup*. Whereas global conformance lookup takes a specialized type and a protocol, local conformance lookup starts from a substitution map, a protocol, and a type parameter for the substitution map's input generic signature.

Indeed, local conformance lookup is the missing piece of the puzzle for understanding the implementation of type substitution with a dependent member type:

1. First, we factor the dependent member type into an associated type declaration together with an abstract conformance. The abstract conformance can be further broken down into a conforming type (the dependent member type's base type) and a protocol (the associated type declaration's parent protocol).

2. Next, we perform a local conformance lookup into the substitution map, with the base type and protocol.

3. Finally, we get the associated type declaration's corresponding type witness from the conformance returned by local conformance lookup.

Local conformance lookup is *compatible* with global conformance lookup, in the following sense: a local conformance lookup with some substitution map, base type and protocol returns the same conformance as first applying the substitution map to the base type, followed by a global conformance lookup with the substituted type and our protocol. This can be expressed in our formalism with the following equation:

$$\begin{aligned}
&\boxed{\text{abstract conformance}} \times \boxed{\text{substitution map}} \\
&= \Big(\boxed{\text{protocol declaration}} \times \boxed{\text{type parameter}}\Big) \times \boxed{\text{substitution map}} \\
&= \boxed{\text{protocol declaration}} \times \Big(\boxed{\text{type parameter}} \times \boxed{\text{substitution map}}\Big) \\
&= \boxed{\text{protocol declaration}} \times \boxed{\text{substituted type}}
\end{aligned}$$

Local conformance lookup is not actually implemented in terms of global conformance lookup, though. Instead, the result is derived directly from the conformances stored in the substitution map itself.

The simplest case is when the abstract conformance directly names a conformance requirement in the substitution map's input generic signature; local conformance lookup returns the corresponding conformance stored in the substitution map. In the general case, local conformance lookup derives the conformance via a *conformance path*. This will be revealed in Chapter 13.

Listing 8.2.: Applying a substitution map to a dependent member type

```
1 struct Concatenation<Elements: Sequence>
2     where Elements.Element: Sequence {
3   typealias InnerIterator = Elements.Element.Iterator
4 }
5
6 // What is the type of 'iter'?
7 let iter: Concatenation<Array<Array<Int>>.InnerIterator = ...
```

Example 8.3. Listing 8.2 shows an example of dependent member type substitution.[2] We're going to work through how the compiler derives the type of the `iter` variable. The type annotation references the `InnerIterator` member type alias with a base type of `Concatenation<Array<Array<Int>>`, so we need to apply the context substitution map of this base type to the underlying type of the type alias declaration.

The generic signature of `Concatenation` is the following:

```
<Elements where Elements: Sequence,
                Elements.[Sequence]Element: Sequence>
```

The context substitution map of `Concatenation<Array<Array<Int>>` is a substitution map for the above input generic signature:

Types
`Elements := Array<Array<Int>>`

Conformances
`Array<Array<Int>>: Sequence`
`Array<Int>: Sequence`

The underlying type of the `InnerIterator` type alias is the bound dependent member type `Elements.[Sequence]Element.[Sequence]Iterator`. To apply our substitution map to this dependent member type, the compiler performs the three steps outlined earlier in this section:

1. The base type of the dependent member type is `Elements.[Sequence]Element`, and the associated type `Iterator` is defined in the `Sequence` protocol. Therefore the abstract conformance is

   `Elements.[Sequence]Element: Sequence`

[2] Are you getting bored of endless variations on `Array<Int>` yet? Feel free to suggest more varied examples!

2. Applying the substitution map to this abstract conformance performs a local conformance lookup into the substitution map. The conforming type and protocol of the abstract conformance is exactly equal to the second conformance requirement in the generic signature, so the local conformance lookup returns the conformance `Array<Int>: Sequence`.

3. The final step projects the type witness for `Iterator` from this conformance. This is a specialized conformance, with the conformance substitution map:

Types		
`Element`	:=	`Int`

   Recall that projecting a type witness from a specialized conformance is defined by first projecting the type witness from the underlying normal conformance, in our case `Array<Element>: Sequence`, and then applying the conformance substitution map, shown above. The type witness for `Iterator` in our normal conformance is an interface type written with respect to the generic signature of `Array`, which is `<Element>`:

   ```
   IndexingIterator<Array<Element>>
   ```

   Applying the conformance substitution map from our specialized conformance to this interface type replaces the `Element` generic parameter with `Int`:

   ```
   IndexingIterator<Array<Int>>
   ```

So the type of `iter` is `IndexingIterator<Array<Int>>`.

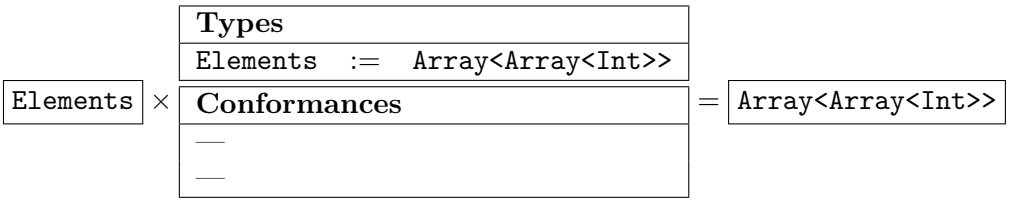
Example 8.4. If you're particularly attentive, you'll remember from Section 7.3 that the construction of the context substitution map of a specialized type is a little tricky, because we have to recursively compute the substituted subject type of each conformance requirement in the generic signature and then perform a global conformance lookup. In the previous example, the generic signature of `Concatenation` has two conformance requirements, and their original and substituted subject types are as follows:

`Elements` ⇒ `Array<Array<Int>>`

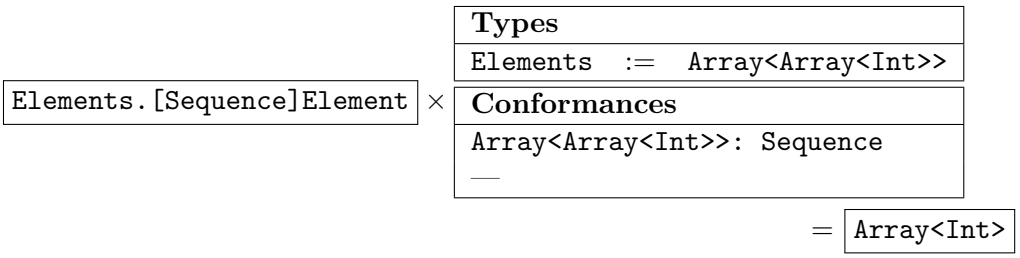
`Elements.[Sequence]Element` ⇒ `Array<Int>`

The computation of each substituted subject type can be understood as applying the *partially-constructed* context substitution map that has been built so far to each original subject type.

For the first subject type, the substitution trivially projects the replacement type of the `Elements` generic parameter:



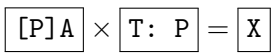
The second time around, the original subject type is itself a dependent member type, so type substitution recursively performs the same dance with a local conformance lookup and type witness projection—if you like, you can work this one out with pen and paper to convince yourself that it is so:



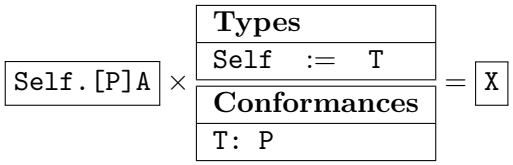
Protocol substitution maps Recall the protocol substitution map construction from Section 7.1, which wraps a conformance `T: P` in a substitution map for the protocol's generic signature `<Self where Self: P>`. Suppose that our protocol `P` declares an associated type `A`, and the type witness for `A` in the conformance `T: P` is some type `X`.

We can now show that the following two are equivalent:

1. Projecting the type witness for `A` from the conformance `T: P`:

2. Applying the protocol substitution map to the declared interface type of `A`, which is the dependent member type `Self.[P]A`:

To see why, we need to recall two facts. First, the dependent member type `Self.[P]A` can be written as the type witness of `A` in the abstract conformance `Self: P`. Second,

applying the protocol substitution map to `Self: P` performs a local conformance lookup which simply projects the original conformance from the substitution map. Therefore, we have:

$$\boxed{\texttt{Self.[P]A}} \times \begin{array}{|l|}\hline \textbf{Types} \\ \hline \texttt{Self := T} \\ \hline \textbf{Conformances} \\ \hline \texttt{T: P} \\ \hline \end{array}$$

$$= \left(\boxed{\texttt{[P]A}} \times \boxed{\texttt{Self: P}} \right) \times \begin{array}{|l|}\hline \textbf{Types} \\ \hline \texttt{Self := T} \\ \hline \textbf{Conformances} \\ \hline \texttt{T: P} \\ \hline \end{array}$$

$$= \boxed{\texttt{[P]A}} \times \left(\boxed{\texttt{Self: P}} \times \begin{array}{|l|}\hline \textbf{Types} \\ \hline \texttt{Self := T} \\ \hline \textbf{Conformances} \\ \hline \texttt{T: P} \\ \hline \end{array} \right)$$

$$= \boxed{\texttt{[P]A}} \times \boxed{\texttt{T: P}}$$

$$= \boxed{\texttt{X}}$$

8.5. Associated Conformances

There is an interesting duality between substitution maps and (normal) conformances, illustrated in Table 8.1.

A substitution map records a replacement type for each generic parameter of a generic signature, and as you saw in the previous section, a normal conformance records a type witness for each associated type of a protocol.

A substitution map also stores a conformance for each conformance requirement in its generic signature. A normal conformance stores an *associated conformance* for each conformance requirement in the protocol's requirement signature.

Recall from Section 6.1 that the printed representation of a requirement signature looks like a generic signature with a single `Self` generic parameter. For example, here is the abridged requirement signature of the standard library's `Collection` protocol:

`<Self where` $\boxed{\texttt{Self: Sequence}}$`,` $\boxed{\texttt{Self.Index: Comparable}}$`, ...>`

The special case of an associated conformance requirement with a subject type of `Self` represents a protocol inheritance relationship, as you already saw in Section 6.1. Other associated conformance requirements constrain the protocol's associated types.

Table 8.1.: Duality between substitution maps and conformances

Substitution map	**Normal conformance**
Input generic signature	Requirement signature
Generic parameter	Associated type declaration
Replacement type	Type witness
Conformance requirement	Associated conformance requirement
Conformance in substitution map	Associated conformance

The conformance checker populates the associated conformance mapping in a normal conformance by computing the substituted subject type of each associated conformance requirement, and then performing a global conformance lookup with this subject type. This is analogous to the conformance lookup performed during the construction of a substitution map (Section 7.3).

The substituted subject type is obtained by applying the protocol substitution map to the subject type of each associated conformance requirement. For example, in the conformance of `Array<Element>` to `Collection`, the substituted subject type of the requirement `Self: Sequence` is just the conforming type:

$$\boxed{\texttt{Self}} \times \left[\begin{array}{|l|}\hline \textbf{Types} \\ \hline \texttt{Self := Array<Element>} \\ \hline \textbf{Conformances} \\ \hline \texttt{Array<Element>: Collection} \\ \hline \end{array}\right] = \boxed{\texttt{Array<Element>}}$$

The substituted subject type of `Self.Index` is the type witness for `Index`, which is `Int`:

$$\boxed{\texttt{Self.Index}} \times \left[\begin{array}{|l|}\hline \textbf{Types} \\ \hline \texttt{Self := Array<Element>} \\ \hline \textbf{Conformances} \\ \hline \texttt{Array<Element>: Collection} \\ \hline \end{array}\right]$$

$$= \boxed{\texttt{[Collection]Index}} \times \boxed{\texttt{Array<Element>: Collection}}$$

$$= \boxed{\texttt{Int}}$$

With the substituted subject types on hand, the conformance checker then performs a global conformance lookup to find each associated conformance:

$$\boxed{\texttt{Sequence}} \times \boxed{\texttt{Array<Element>}} = \boxed{\texttt{Array<Element>: Sequence}}$$

$$\boxed{\texttt{Comparable}} \times \boxed{\texttt{Int}} = \boxed{\texttt{Int: Comparable}}$$

Notation We're going to use the notation `(Self.Index: Comparable)` for associated conformance requirements. The parentheses will serve as a visual reminder that they are different from abstract conformances, which use the notation `T: P`. The distinction is important; an abstract conformance describes a type parameter that is known to conform to a protocol in some *generic* signature (possibly as a non-trivial consequence of other requirements), whereas an associated conformance requirement is a *specific* requirement directly appearing in a protocol's *requirement* signature.

Projection Projecting an associated conformance from a normal conformance can be understood as the action of an associated conformance requirement (from a protocol's requirement signature) on the left of a normal conformance (to this protocol):

$$\boxed{\text{conformance requirement}} \times \boxed{\text{normal conformance}} = \boxed{\text{associated conformance}}$$

With a specialized conformance, we do the same thing as when getting a type witness; first, we get the associated conformance from the underlying normal conformance, and then we apply the conformance substitution map:

$$\begin{aligned}
&\boxed{\text{conformance requirement}} \times \boxed{\text{specialized conformance}} \\
&= \boxed{\text{conformance requirement}} \times \left(\boxed{\text{normal conformance}} \times \boxed{\text{substitution map}} \right) \\
&= \left(\boxed{\text{conformance requirement}} \times \boxed{\text{normal conformance}} \right) \times \boxed{\text{substitution map}} \\
&= \boxed{\text{associated conformance}} \times \boxed{\text{substitution map}}
\end{aligned}$$

Now we can project associated conformances from normal conformances and specialized conformances. Last but not least, we need to define associated conformance projection from an abstract conformance. Just as the type witnesses of an abstract conformance are dependent member types, associated conformances of an abstract conformance are other abstract conformances:

$$\boxed{\texttt{(Self.[P]A: Q)}} \times \boxed{\texttt{T: P}} = \boxed{\texttt{T.[P]A : Q}}$$

Example 8.5. The associated conformances of a normal conformance can themselves be any kind of conformance, including normal, specialized or abstract. Listing 8.3 shows these possibilities. The protocol `P` states three associated conformance requirements, and each of the associated conformances of the normal conformance `S<T>: P` are a different kind of conformance:

Requirement	Associated conformance	Kind
`A: Equatable`	`Int: Equatable`	Normal
`B: Equatable`	`Array<Int>: Equatable`	Specialized
`C: Equatable`	`T: Equatable`	Abstract

Listing 8.3.: Different kinds of associated conformances

```
protocol P {
  associatedtype A: Equatable
  associatedtype B: Equatable
  associatedtype C: Equatable
}

struct S<T: Equatable>: P {
  typealias A = Int
  typealias B = Array<Int>
  typealias C = T
}
```

The case where the associated conformance is abstract is important, because it arises when the type witness is a type parameter of the conforming type's generic signature.

Now consider what happens when we project the associated conformance `(C: P)` from the specialized conformance `S<String>: P`:

$$\boxed{\texttt{(C: P)}} \times \boxed{\texttt{S<String>: P}}$$

$$= \left(\boxed{\texttt{(C: P)}} \times \boxed{\texttt{S<T>: P}} \right) \times \begin{array}{|l|}\hline \textbf{Types} \\ \hline \texttt{T := String} \\ \hline \textbf{Conformances} \\ \hline \texttt{String: Equatable} \\ \hline \end{array}$$

$$= \boxed{\texttt{T: Equatable}} \times \begin{array}{|l|}\hline \textbf{Types} \\ \hline \texttt{T := String} \\ \hline \textbf{Conformances} \\ \hline \texttt{String: Equatable} \\ \hline \end{array}$$

The associated conformance projection operation actually turns around and reduces to a local conformance lookup into the substitution map, which gives us the final result:

$$\boxed{\texttt{T: Equatable}} \times \begin{array}{|l|}\hline \textbf{Types} \\ \hline \texttt{T := String} \\ \hline \textbf{Conformances} \\ \hline \texttt{String: Equatable} \\ \hline \end{array} = \boxed{\texttt{String: Equatable}}$$

This has some unexpected consequences, which are explored in Section 13.1.

8.6. Source Code Reference

Key source files:

- include/swift/AST/ProtocolConformanceRef.h
- include/swift/AST/ProtocolConformance.h
- lib/AST/ProtocolConformanceRef.cpp
- lib/AST/ProtocolConformance.cpp

Other source files:

- include/swift/AST/DeclContext.h
- include/swift/AST/Module.h
- lib/AST/ConformanceLookupTable.h
- lib/AST/ConformanceLookupTable.cpp
- lib/AST/Module.cpp

`IterableDeclContext` *class*

Base class inherited by `NominalTypeDecl` and `ExtensionDecl`.

- `getLocalConformances()` returns a list of conformances directly declared on this nominal type or extension.

`NominalTypeDecl` *class*

See also Section 4.4.

- `getAllConformances()` returns a list of all conformances declared on this nominal type, its extensions, and inherited from its superclass, if any.

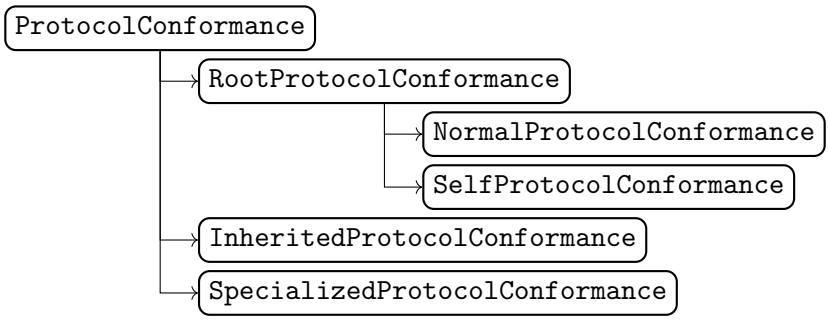
`ConformanceLookupTable` *class*

A conformance lookup table for a nominal type. Every `NominalTypeDecl` has a private instance of this class, but it is not exposed outside of the global conformance lookup implementation.

`ModuleDecl` *class*

See also Section 2.6.

Figure 8.3.: The `ProtocolConformance` class hierarchy



- `lookupConformance()` returns the conformance of a type to a protocol. This is the a global conformance lookup operation.

`ProtocolConformanceRef` *class*

A protocol conformance. Stores a single pointer, and is cheap to pass around by value.

- `isInvalid()` answers if this is an invalid conformance reference, meaning the type did not actually conform to the protocol.
- `isAbstract()` answers if this is an abstract conformance reference.
- `isConcrete()` answers if this is a concrete conformance reference.
- `getConcrete()` returns the `ProtocolConformance` instance if this is a concrete conformance.
- `getRequirement()` returns the `ProtocolDecl` instance if this is an abstract or concrete conformance.
- `subst()` returns the protocol conformance obtained by applying a substitution map to this conformance.

`ProtocolConformance` *class*

A concrete protocol conformance. This class is the root of a class hierarchy shown in Figure 8.3. Concrete protocol conformances are allocated in the AST context, and are always passed by pointer.

- `getType()` returns the conforming type.
- `getProtocol()` returns the conformed protocol.

- `getTypeWitness()` returns the type witness for an associated type.
- `getAssociatedConformance()` returns the associated conformance for a conformance requirement in the protocol's requirement signature.
- `subst()` returns the protocol conformance obtained by applying a substitution map to this conformance.

`RootProtocolConformance` *class*

Abstract base class for `NormalProtocolConformance` and `SelfProtocolConformance`. Inherits from `ProtocolConformance`.

`NormalProtocolConformance` *class*

A normal protocol conformance. Inherits from `RootProtocolConformance`.

- `getDeclContext()` returns the conforming declaration context, either a nominal type declaration or extension.
- `getGenericSignature()` returns the generic signature of the conforming context.
- `finishSignatureConformances()` computes the associated conformances of this conformance. Not intended to be called directly.

`SpecializedProtocolConformance` *class*

A specialized protocol conformance. Inherits from `ProtocolConformance`.

- `getGenericConformance()` returns the underlying normal conformance.
- `getSubstitutionMap()` returns the conformance substitution map.

9. Generic Environments

In Chapter 3, type parameters and archetypes were introduced as two kinds of "abstract types." So far, we've only talked about type parameters, which appear in the interface types of declarations. Archetypes appear in the types of expressions inferred by the expression type checker, and in the SIL instructions constructed by lowering expressions to SIL.

To understand how archetypes are different from type parameters, consider two key properties of type parameters:

1. Type parameters only have meaning with respect to their generic signature. For example, generic signature queries (Section 6.4) are called with a generic signature together with a type parameter.

2. In a generic signature, two type parameters that are not canonical-equal might still belong to the same equivalence class, and be reduced-equal. Type parameters can represent the different "spellings" which are equivalent as a result of same-type requirements. This gives the two levels of equality on interface types: canonical equality, and reduced equality with respect to a generic signature.

An archetype represents a reduced type parameter in a specific *generic environment*. Unlike type parameters, they are self-describing, since they point back at their parent generic environment. The underlying type parameter of an archetype is always reduced, so an equivalence class of type parameters is represented by a single archetype in a given generic environment.

Recall that a type containing type parameters is called an interface type. Similarly, a type containing archetypes is called a *contextual type*. A pair of operations define a mapping between interface types and contextual types:

- **Mapping into an environment** transforms an interface type into a contextual type by replacing the interface type's type parameters with archetypes. Any type parameters that are not reduced are replaced by their reduced type first. This mapping is performed with respect to a fixed generic environment.

- **Mapping out of an environment** transforms a contextual type into an interface type by replacing each archetype with the reduced type parameter it represents. This operation does not take a generic environment; all archetypes know their interface type.

There are three kinds of generic environment:

- Every generic signature has exactly one **primary generic environment**. The archetypes in the primary environment are called *primary archetypes*. Primary archetypes represent the generic parameters of a function inside of a function body, both in AST statement and expression nodes, and in the SIL instructions of a SIL function. Primary generic environments preserve the sugared names of generic parameters for the printed representation of an archetype, so two canonically-equal but not pointer-equal generic signatures will instantiate distinct primary generic environments.

  $$\boxed{\text{generic signature}} \Leftrightarrow \boxed{\text{primary environment}}$$

- When a declaration has an opaque return type, an **opaque generic environment** is created for each unique substitution map of the declaration's generic signature. The archetypes of this environment are used for both the declaration and references to the declaration's opaque result type. These are discussed in Chapter 14.

  $$\boxed{\text{opaque type declaration}} \times \boxed{\text{substitution map}} = \boxed{\text{opaque environment}}$$

- An **opened generic environment** is created when an existential value is opened inside an expression. Opened archetypes represent the concrete payload of a value of existential type. A call site where an existential value is opened will instantiate a unique opened generic environment, and the usage of the opened archetypes is scoped to the call's argument expressions. Opened archetypes are discussed in Chapter 15.

  $$\boxed{\text{generic signature}} \times \boxed{\text{existential type}} \times \boxed{\text{UUID}} = \boxed{\text{opened environment}}$$

A generic environment contains a lazily-populated mapping from the reduced type parameters of its generic signature to archetypes. The archetypes instantiated by a generic environment are not pointer-equal or canonical-equal to the archetypes of any other generic environment.

Motivation You might wonder why archetypes exist at all, when at first glance, they appear equivalent to a reduced type parameter together with a generic signature. In the case of primary archetypes at least, the reason is partly historical. However, the additional indirection provided by creating multiple generic environments from a single generic signature allows archetypes to represent abstract types which are not described by the generic parameters that are in the scope of a generic declaration, namely opaque return types and existential types.

Local requirements The *local requirements* of an archetype describe the behavior of the archetype's underlying type parameter in the generic signature of the archetype's generic environment. Local requirements are stored inside the archetype. They are derived when the archetype is first constructed within a generic environment using the generic signature queries of Section 6.4:

- **Required protocols:** a minimal and canonical list of protocols the archetype is known to conform to, from the `getRequiredProtocols()` generic signature query.
- **Superclass bound:** an optional superclass type that the archetype is known to be a subclass of, computed by mapping the interface type returned by the `getSuperclassBound()` generic signature query into the generic environment.
- **Requires class flag:** a boolean indicating if the archetype is class-constrained, computed from the `requiresClass()` generic signature query.
- **Layout constraint:** an optional layout constraint the archetype is known to satisfy, computed from the `getLayoutConstraint()` generic signature query.

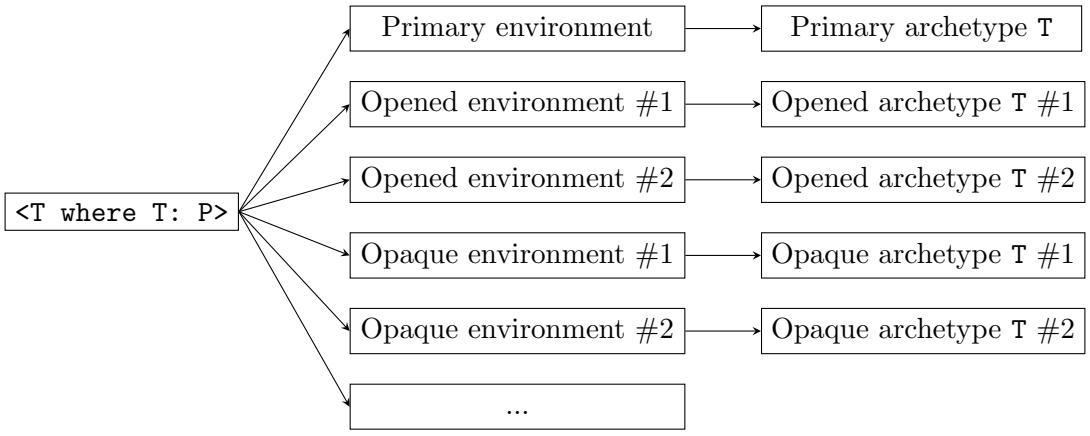
There is no equivalent of the `getConcreteType()` generic signature query in the world of archetypes. Archetypes represent reduced type parameters, and type parameters fixed to a concrete type are not reduced. If a generic signature fixes a type parameter to a concrete type, mapping the type parameter into an environment will first replace the type parameter with its concrete type, and then recursively map the resulting concrete type into the environment. If the concrete type contains type parameters, they will be replaced with archetypes (or concrete types, if they are themselves fixed to concrete types).

For the same reason, generic signature queries to operate on reduced types do not have equivalents in the world of archetypes. Reduced types are computed as part of mapping an interface type into a generic environment. Since archetypes represent reduced type parameters, the three notions of pointer, canonical and reduced equality collapse into one. Contextual types that contain archetypes may still differ by type sugar in other positions, however canonical equality is sufficient to determine if two contextual types represent the same reduced type.

Global conformance lookup In Section 8.1, we introduced global conformance lookup on nominal types. It generalizes to archetypes in a straightforward way:

1. If the archetype conforms abstractly via a protocol conformance requirement, global conformance lookup returns an abstract conformance.
2. If the archetype conforms concretely via a superclass requirement, global conformance lookup recursively calls itself with the archetype's superclass type and returns an inherited conformance (Section 16.1).

Figure 9.1.: A generic signature with multiple generic environments



Qualified name lookup Continuing the trend of operations on concrete types that also support archetypes, an archetype can be used as the base type of a qualified name lookup. Recall the notion of a reachable declaration context from Section 7.1. The reachable declaration contexts of an archetype are the protocols it conforms to, the class declaration of its superclass type, and any protocols the superclass conforms to.

Context substitution map An archetype can serve as the base type when computing a context substitution map for a declaration context. The declaration context can either be a protocol context or a class context. In the case of a protocol context, the archetype can conform abstractly or concretely, as described above; a protocol substitution map is constructed from the archetype and the conformance returned by global conformance lookup. The case where the declaration context is a class is handled by a small addition to Algorithm 16.1. Before proceeding with the main algorithm, we first check if the type `T` is an archetype, and replace it with the archetype's superclass type. The class context must be the class declaration (or an extension) of some ancestor class of the archetype's superclass requirement.

Invariants It is unwise to mix interface types and contextual types. Generally, when talking about the external interface of a declaration, you should use interface types, and when talking about types appearing inside the body of a function, you should use contextual types. A pair of recursively-computed properties distinguish interface types from archetypes:

`hasTypeParameter()` answers if the type contains a type parameter.

`hasArchetype()` answers if the type contains a primary or opened archetype. Types containing opaque archetypes do not respond with `true` to this call, for reasons that are explained later.

These predicates should be used in assertions to establish invariants. Generally, the predicate you assert is the negation of the *opposite* predicate. If your function only operations on interface types, you should check for the absence of archetypes; if your function only expects contextual types, you should check for the absence of type parameters. This allows for fully-concrete types, which contain neither type parameters nor archetypes.

Mapping a type into an environment asserts that the input type does not contain archetypes, and similarly mapping a type out of an environment asserts that the input type does not contain type parameters. This means you cannot call these operations "just in case"; you need to establish that you're dealing with the correct kind of type upfront with an additional check or assertion. Furthermore, mapping a type out of an environment asserts that the type does not contain opened archetypes. Since the type parameter of an opened archetype does not correspond to a type parameter in the declaration's generic signature, mapping an opened archetype out of its environment is not a meaningful operation.

9.1. Primary Archetypes

Every generic signature stores its primary generic environment. The archetypes of a primary generic environment are called *primary archetypes*.

$$\boxed{\text{reduced type parameter}} \times \boxed{\text{generic signature}} = \boxed{\text{primary archetype}}$$

Example 9.1. Consider this generic function:

```
1 func sum<S: Sequence<Int>>(_ seq: S) -> Int {
2   ...
3 }
```

The function's generic signature:

```
<S where S: Sequence, S.[Sequence]Element == Int>
```

We can write four type parameters for this generic signature:

```
S
S.[Sequence]Element
S.[Sequence]Iterator
S.[Sequence]Iterator.[IteratorProtocol]Element
```

The type parameters `S` and `S.[Sequence]Iterator` are reduced, so they map to two distinct archetypes ⟦`S`⟧ and ⟦`S.[Sequence]Iterator`⟧ in the function's primary generic environment. The other two type parameters not reduced, because they are fixed to the concrete type `Int`. Mapping them into the environment produces `Int`.

Contextual types are substitutable in the same way as interface types. Applying a substitution map to a contextual type is defined by first mapping the contextual type out of its environment.

$$\boxed{\text{contextual type}} \times \boxed{\text{substitution map}} = \boxed{\text{interface type}} \times \boxed{\text{substitution map}}$$

Thus, there is no distinction between substitution maps operating on interface types and contextual types. However, there is a distinction when you look at the replacement types that are *output* by the substitution. We can define an *interface substitution map* as one where the replacement types are interface types, and a *contextual substitution map* as one where the replacement types are contextual types. Applying an interface substitution map to an interface type or contextual type always produces an interface type. Applying a contextual substitution map to an interface type or contextual type always produces a contextual type:

$$\boxed{\text{interface type}} \times \boxed{\text{interface substitution map}} = \boxed{\text{substituted interface type}}$$
$$\boxed{\text{contextual type}} \times \boxed{\text{interface substitution map}} = \boxed{\text{substituted interface type}}$$
$$\boxed{\text{interface type}} \times \boxed{\text{contextual substitution map}} = \boxed{\text{substituted contextual type}}$$
$$\boxed{\text{contextual type}} \times \boxed{\text{contextual substitution map}} = \boxed{\text{substituted contextual type}}$$

We saw in Section 7.2 that every generic signature has an identity substitution map, and applying the identity substitution map to an interface type leaves the type unchanged:

$$\boxed{\text{interface type}} \times \boxed{\text{identity substitution map}} = \boxed{\text{interface type}}$$

Every generic environment has a *forwarding substitution map* that replaces each generic parameter with the generic parameter mapped into the environment. The forwarding substitution map plays the role of an identity with contextual types. Applying the forwarding substitution map to a contextual type leaves the type unchanged:

$$\boxed{\text{contextual type}} \times \boxed{\text{forwarding substitution map}} = \boxed{\text{contextual type}}$$

What happens if you apply the identity substitution map to a *contextual* type? Applying *any* substitution map to a contextual type first maps it out of its environment, producing an interface type, and the identity substitution map leaves all type parameters in this interface type unchanged. Thus, applying the identity substitution map to a contextual type is the same as mapping the contextual type out of its environment:

$$\boxed{\text{contextual type}} \times \boxed{\text{identity substitution map}} = \boxed{\text{interface type}}$$

There is one final combination. Applying the forwarding substitution map to an interface type replaces all type parameters with archetypes, so it is the same operation as mapping the interface type into the environment:

$$\boxed{\text{interface type}} \times \boxed{\text{forwarding substitution map}} = \boxed{\text{contextual type}}$$

The replacement types of a substitution map can be mapped into an environment by applying the forwarding substitution map for the appropriate generic environment on the right:

$$\boxed{\text{interface substitution map}} \times \boxed{\text{forwarding substitution map}} = \boxed{\text{contextual substitution map}}$$

Applying an interface substitution map and then mapping the result into an environment has the same effect has applying the corresponding contextual substitution map:

$$\left(\boxed{\text{interface type}} \times \boxed{\text{interface substitution map}}\right) \times \boxed{\text{generic environment}}$$
$$= \boxed{\text{interface type}} \times \left(\boxed{\text{interface substitution map}} \times \boxed{\text{generic environment}}\right)$$
$$= \boxed{\text{interface type}} \times \boxed{\text{contextual substitution map}}$$

Another way to map the replacement types of a contextual substitution map out of their environment is to apply the identity substitution map on the right. However, this requires finding the output generic signature for the substitution map. Just as contextual types can be mapped out of an environment without providing the environment, substitution maps support a **map replacement types out of environment** operation.

Archetypes are not "inherited" There's a potential pitfall worth mentioning. Recall that when generic declarations nest, the inner declaration inherits the generic parameters and requirements of the outer declaration, possibly adding new generic parameters or requirements:

```
1  func myAlgorithm<S: Sequence>(_ seq: S) where S.Element: Comparable {
2    func helper<T: Sequence>(_ t: T) where T.Element == S {
3      let s1: S = ...
4      print(s1)
5    }
6
7    let s2: [S] = [seq]
8    helper(s2)
9  }
```

The inner `helper()` function has a distinct generic signature, and therefore a distinct generic environment, from the outer `complexAlgorithm()` function. In particular, the outer function's generic parameter `S` maps to two *different* archetypes inside the two declarations; say, $[\![\texttt{S}]\!]_1$ and $[\![\texttt{S}]\!]_2$. The type of the expression `s1` in `print(s1)` is $[\![\texttt{S}]\!]_1$, and the type of `s2` in `helper(s2)` is $[\![\texttt{S}]\!]_2$. The call to `helper()` supplies a substitution map which replaces the generic parameter `S` with the archetype $[\![\texttt{S}]\!]_2$, and `T` with the contextual type `Array<`$[\![\texttt{S}]\!]_2$`>`.

The only case where a generic environment is inherited by an inner declaration is if the inner declaration is not "more generic" in any way; it does not declare generic parameters, *or* a `where` clause. As another example, anonymous closure expressions always inherit the generic environment of the outer declaration, because they cannot be generic except by referencing outer generic parameters.

When implementing type checker logic for nested function declarations, take care to map types into the correct generic environment, corresponding to the exact declaration where they will be used.

9.2. Source Code Reference

`GenericEnvironment` *class*

A generic environment. Instances are allocated in the AST context, and passed by pointer.

- `getGenericSignature()` returns this generic environment's generic signature.
- `mapTypeIntoContext()` returns the contextual type obtained by mapping an interface type into this generic environment.
- `getForwardingSubstitutionMap()` returns a substitution map for mapping each generic parameter to its contextual type—an archetype, or a concrete type if the generic parameter is fixed to a concrete type via a same-type requirement.

`GenericSignature` *class*

See also Section 6.5.

- `getGenericEnvironment()` returns the primary generic environment associated with this generic signature.

`TypeBase` *class*

See also Section 3.6.

- `mapTypeOutOfContext()` returns the interface type obtained by mapping this contextual type out of its generic environment.

`SubstitutionMap` *class*

See also Section 7.5.

- `mapReplacementTypesOutOfContext()` returns the substitution map obtained by mapping this substitution map's replacement types and conformances out of their generic environment.

`ProtocolConformanceRef` *class*

See also Section 8.6.

- `mapConformanceOutOfContext()` returns the protocol conformance obtained by mapping this protocol conformance out of its generic environment.

`DeclContext` *class*

See also Section 4.4.

- `getGenericEnvironmentOfContext()` returns the generic environment of the innermost generic declaration containing this declaration context.
- `mapTypeIntoContext()` Maps an interface type into the primary generic environment for the innermost generic declaration. If at least one outer declaration context is generic, this is equivalent to:

```
1 dc->getGenericEnvironmentOfContext()->mapTypeIntoContext(type);
```

  For convenience, the `DeclContext` version of `mapTypeIntoContext()` also handles the case where no outer declaration is generic. In this case, it returns the input type unchanged, after asserting that it does not contain any type parameters (since type parameters appearing outside of a generic declaration are nonsensical).

Part II.

Odds and Ends

10. Type Resolution

10.1. Identifier Type Representations

10.2. Checking Generic Arguments

10.3. Protocol Type Aliases

10.4. Source Code Reference

11. Building Generic Signatures

11.1. Requirement Inference

11.2. Desugared Requirements

11.3. Minimal Requirements

11.4. Source Code Reference

12. Extensions

12.1. Constrained Extensions

12.2. Conditional Conformances

12.3. Source Code Reference

13. Conformance Paths

13.1. Recursive Conformances

14. Opaque Return Types

14.1. Opaque Archetypes

14.2. Referencing Opaque Archetypes

15. Existential Types

15.1. Opened Existentials

15.2. Self-Conforming Protocols

16. Class Inheritance

Algorithm 16.1 (Iterated superclass type)**.** As input, takes a class type `T` and a superclass declaration `D`. Returns the superclass type of `T` for `D`.

1. Let `C` be the class declaration referenced by `T`. If `C` = `D`, return `T`.
2. If `C` does not have a superclass type, fail with an invariant violation; `D` is not actually a superclass of `T`.
3. Otherwise, apply the context substitution map of `T` to the superclass type of `C`. Assign this new type to `T`, and go back to Step 1.

16.1. Inherited Conformances

16.2. Override Checking

17. Witness Thunks

Part III.

The Requirement Machine

18. Property Map

Bibliography

[1] N. Cook, N. Chandler, and M. Ricketson, "SE-0281: `@main`: Type-based program entry points," 2020. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0281-main-attribute.md

[2] "Swift intermediate language (SIL)," 2016. [Online]. Available: https://github.com/apple/swift/blob/main/docs/SIL.rst

[3] A. Zhilin, "SE-0077: Improved operator declarations," 2016. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0077-operator-precedence.md

[4] T. Allevato and D. Gregor, "SE-0091: Improving operator requirements in protocols," 2016. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0091-improving-operators-in-protocols.md

[5] D. Gregor, "Request evaluator," 2018. [Online]. Available: https://github.com/apple/swift/blob/main/docs/RequestEvaluator.md

[6] J. Rose, "Dependency analysis," 2015. [Online]. Available: https://github.com/apple/swift/blob/main/docs/DependencyAnalysis.md

[7] S. Pestov, "SE-0193: Cross-module inlining and specialization," 2018. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0193-cross-module-inlining-and-specialization.md

[8] J. Rose and B. Cohen, "SE-0260: Library evolution for stable ABIs," 2019. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0260-library-evolution.md

[9] J. McCall and D. Gregor, "SE-0296: Async/await," 2020. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0296-async-await.md

[10] D. Gregor, "SE-0021: Naming functions with argument labels," 2016. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0021-generalized-naming.md

[11] A. Zheng, "SE-0111: Remove type system significance of function argument labels," 2016. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0111-remove-arg-label-type-significance.md

[12] C. Lattner, "SE-0029: Remove implicit tuple splat behavior from function applications," 2016. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0029-remove-implicit-tuple-splat.md

[13] ——, "SE-0066: Standardize function type argument syntax to require parentheses," 2016. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0066-standardize-function-type-syntax.md

[14] V. S. and A. Zheng, "SE-0110: Distinguish between single-tuple and multiple-argument function types," 2016. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0110-distinguish-single-tuple-arg.md

[15] P. Yaskevich, H. Borla, and S. Pestov, "SE-0346: Lightweight same-type requirements for primary associated types," 2022. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0346-light-weight-same-type-syntax.md

[16] H. Borla, "SE-0335: Introduce existential `any`," 2021. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0335-existential-any.md

[17] F. Kellison-Linn, "SE-0315: Type placeholders," 2021. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0315-placeholder-types.md

[18] C. Lattner, "SE-0048: Generic type aliases," 2016. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0048-generic-typealias.md

[19] C. Eidhof, "SE-0148: Generic subscripts," 2017. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0148-generic-subscripts.md

[20] D. Hart and A. Zheng, "SE-0156: Class and subtype existentials," 2017. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0156-subclass-existentials.md

[21] D. Hart, R. Widman, and P. Jahkola, "SE-0081: Move `where` clause to end of declaration," 2016. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0081-move-where-expression.md

[22] A. Latsis, "SE-0261: `where` clauses on contextually generic declarations," 2019. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0267-where-on-contextually-generic.md

[23] D. Gregor, "SE-0341: Opaque parameter declarations," 2022. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0341-opaque-parameters.md

[24] D. Hart, J. Bandes-Storch, and D. Gregor, "SE-0142: Permit where clauses to constrain associated types," 2017. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0142-associated-types-constraints.md

[25] D. Gregor, E. Sadun, and A. Zheng, "SE-0157: Support recursive constraints on associated types," 2017. [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0157-recursive-protocol-constraints.md

Index