Generic Signatures

A generic signature describes a set of generic type parameters along with a set of constraints on those parameters. Generic entities in Swift have a corresponding generic signature. For example, the following generic function:

```
func foo<C1: Collection, C2: Collection>(c1: C1, c2: C2)
  where C1.Element: Equatable, C1.Element == C2.Element
{ }
```

has the generic signature:

```
<C1, C2 where C1: Collection, C2: Collection, C1.Element: Equatable,
C1.Element == C2.Element>
```

Generic signatures are used in a few places within the ABI, including:

- The mangled names of generic entities include the generic signature
- The generic type parameters and protocol-conformance constraints in a generic signature are mapped to type metadata and witness-table parameters in a generic function, respectively.
- The entries in a protocol witness table correspond to a variant of the generic signature of a protocol called the requirement signature.

Whenever used in the ABI, a generic signature must be both *minimal* and *canonical*, as defined below.

Throughout this document, "type parameter" is used to refer to either a generic type parameter (such as `C1` in the example above) or a nested type rooted at a generic type parameter (such as `C1.Element` or `C1.Iterator.Element`).

Minimization

A generic constraint is considered *redundant* if it can be proven true based on some combination of other constraints within the same generic signature. Redundant constraints can be removed from a generic signature without affecting the semantics of the signature. A generic signature is *minimal* when it does not contain any constraints that are redundant.

Consider the following generic signature:

```
<C1, C2 where C1: Collection, C2: Collection, C1.Element: Equatable,
 C1.Element == C2.Element, C2.Element: Equatable>
```

The constraint `C1.Element: Equatable` is redundant (because it can be proven based on `C1.Element == C2.Element` and `C2.Element: Equatable`). Similarly, `C2.Element: Equatable` is redundant (based on `C1.Element == C2.Element` and `C1.Element: Equatable`). Either one of these constraints can be removed without changing the semantics of the generic signature, and the resulting generic signature will be minimal (there are no redundant constraints that

remain). As such, there are two minimal generic signatures that describe this set of constraints:

```
<C1, C2 where C1: Collection, C2: Collection, C1.Element: Equatable,
 C1.Element == C2.Element>
```

and

```
<C1, C2 where C1: Collection, C2: Collection, C1.Element == C2.Element,
 C2.Element: Equatable>
```

Removing both constraints would produce a semantically different generic signature. The following section on canonicalization details why the first generic signature is the signature used for ABI purposes.

Canonicalization

A generic signature is *canonical* when each of its constraints is canonical and the entries in the generic signature appear in canonical order.

1. Generic type parameters are listed first ordered by type parameter ordering
2. Constraints follow, ordered first by the type parameter ordering of the left-hand operand and then by constraint kind. The left-hand side of a constraint is always a type parameter (call it `T`). Constraints are ordered as follows:
    1. A superclass constraint `T: C`, where `C` is a class.
    2. A layout constraint (e.g., `T: some-layout`), where the right-hand side is `AnyObject` or one of the non-user-visible layout constraints like `_Trivial`.
    3. Conformance constraints `T: P`, where `P` is a protocol. The conformance constraints for a given type parameter `T` are further sorted using the protocol ordering.
    4. A same-type constraint `T == U`, where `U` is either a type parameter or a concrete type.

Type parameter ordering

Given two type parameters `T1` and `T2`, `T1` precedes `T2` in the canonical ordering if:

- `T1` and `T2` are generic type parameters with depths `d1` and `d2`, and indices `i1` and `i2`, respectively, and either `d1 < d2` or `d1 == d2` and `i1 < i2`;
- `T1` is a generic type parameter and `T2` is a nested type `U2.A2`; or
- `T1` is a nested type `U1.A1` and `T2` is a nested type `U2.A2`, where `A1` and `A2` are associated types of the protocols `P1` and `P2`, respectively, and either
    - `U1` precedes `U2` in the canonical ordering, or
    - `U1 == U2` and the name of `A1` lexicographically precedes the name of `A2`, or

  - `A1` is a *root* associated type (defined below) and `A2` is not a root associated type
  - `U1 == U2` and `P1` precedes `P2` in the canonical ordering defined by the following section on protocol ordering.

A *root* associated type is an associated type that first declares that associated type name within a protocol hierarchy. An inheriting protocol may declare an associated type with the same name, but that associated type is not a root. For example:

```
protocol P {
  associatedtype A  // root associated type
}

protocol Q: P {
  associatedtype B   // not a root associated type ("overrides" P.A)
  associatedtype C   // root associated type
}
```

Protocol ordering

Given two protocols `P1` and `P2`, protocol `P1` precedes `P2` in the canonical ordering if:

- `P1` is in a different module than `P2` and the module name of `P1` lexicographically precedes the module name of `P2`, or
- `P1` and `P2` are in the same module and the name of `P1` lexicographically precedes the name of `P2`.

Canonical constraints

A given constraint can be described in multiple ways. In our running example, the conformance constraint for the element type can be expressed as either `C1.Element: Equatable` or `C2.Element: Equatable`, because `C1.Element` and `C2.Element` name the same type. There might be an infinite number of ways to name the same type (e.g., `C1.SubSequence.SubSequence.Iterator.Element` is also equivalent to `C1.Element`). All of the spellings that refer to the same time comprise the *equivalence class* of that type.

Each equivalence class has a corresponding *anchor*, which is a type parameter that is the least type according to the type parameter ordering. Anchors are used to describe requirements canonically. A concrete type (i.e., a type that is not a type parameter) is canonical when each type parameter within it has either been replaced with its equivalent (canonical) concrete type (when such a constraint exists in the generic signature) or is the anchor of its equivalence class.

A layout or conformance constraint is canonical when its left-hand side is the anchor of its equivalence class. A superclass constraint is canonical when its

left-hand side is the anchor of its equivalence class and its right-hand side is a canonical concrete (class) type. Same-type constraint canonicalization is discussed in detail in the following section, but some basic rules apply: the left-hand side is always a type parameter, and the right-hand side is either a type parameter that follows the left-hand side (according to the type parameter ordering) or is a canonical concrete type.

Same-type constraints

The canonical form of superclass, layout, and conformance constraints are directly canonicalizable once the equivalence classes are known. Same-type constraints, on the other hand, are responsible for forming those equivalence classes. Let's expand our running example to include a third `Collection`:

```
<C1, C2, C3 where C1: Collection, C2: Collection, C3: Collection,
 C1.Element: Equatable, C1.Element == C2.Element, C1.Element == C3.Element>
```

All of `C1.Element`, `C2.Element`, and `C3.Element` are in the same equivalence class, which can be formed by different sets of same-type constraints, e.g.,

```
C1.Element == C2.Element, C1.Element == C3.Element
```

or

```
C1.Element == C2.Element, C2.Element == C3.Element
```

or

```
C1.Element == C3.Element, C2.Element == C3.Element
```

All of these sets of constraints have the same effect (i.e., form the same equivalence class), but the second one happens to be the canonical form.

The canonical form is determined by first dividing all of the types in the same equivalence class into distinct components. Two types `T1` and `T2` are in the same component if the same type constraint `T1 == T2` can be proven true based on other known constraints in the generic signature (i.e., if `T1 == T2` would be redundant). For example, `C1.Element` and `C1.SubSequence.Element` are in the same component, because `C1: Collection` and the `Collection` protocol contains the constraint `Element == SubSequence.Element`. However, `C1.Element` and `C2.Element` are in different components.

Each component has a *local anchor*, which is a type parameter that is the least type within that component, according to the type parameter ordering. The local anchors are then sorted (again, using type parameter ordering); call the anchors `A1`, `A2`, ..., `An` where `Ai < Aj` for `i < j`. The canonical set of constraints depends on whether the equivalence class has been constrained to a concrete type:

- If there exists a same-type constraint `T == C`, where `T` is a member of the equivalence class and `C` is a concrete type, the set of same-type constraints

for the equivalence class is `A1 == C`, `A2 == C`, ..., `An == C`.

- If there is no such same-type constraint, the set of canonical same-type constraints for the equivalence class is `A1 == A2`, `A2 == A3`, ..., `A(n-1) == An`.

The second case is illustrated above; note that it requires `n-1` same-type constraints to form an equivalence class with `n` separate components.

For the first case, consider a function that operates on `String` collections:

```
func manyStrings<C1: Collection, C2: Collection, C3: Collection>(
       c1: C1, c2: C2, c3: C3)
  where C1.Element == String, C1.Element == C2.Element,
        C1.Element == C3.SubSequence.Element
{ }
```

The minimal canonical generic signature for this function is:

```
<C1, C2, C3 where C1: Collection, C2: Collection, C3: Collection,
 C1.Element == String, C2.Element == String, C3.Element == String>
```

Note that `C1.Element`, `C2.Element`, and `C3.SubSequence.Element` are all in the same equivalence class, but are in different components. The first two are the local anchors of their respective components, while the local anchor for the third component is `C3.Element`. Because the equivalence class is constrained to a concrete type (`String`), the canonical form includes a same-type constraint making each local anchor equivalent to that concrete type.

Requirement signatures

A protocol can introduce a number of constraints, including inherited protocols and constraints on associated types. The *requirement signature* of a protocol is a form of a generic signature that describes those constraints. For example, consider a `Collection` protocol similar to the one in the standard library:

```
protocol Collection: Sequence where SubSequence: Collection {
  associatedtype Index
  associatedtype Indices: Collection where Indices.Element == Index
  // ...
}
```

This protocol introduces a number of constraints: `Self: Sequence` (stated as inheritance), `Self.SubSequence: Collection` (in the protocol where clause), `Self.Indices: Collection` (associated type declaration) and `Self.Indices.Element == Index` (associated type where clause). These constraints, which are directly implied by `Self: Collection`, form the *requirement signature* of a protocol. As with other generic signatures used for the ABI, the requirement signature is minimal and canonical according to the rules described elsewhere in this document.