(Source: https://lists.swift.org/pipermail/swift-evolution/Week-of-Mon-20160229/011666.html, authored by Douglas Gregor)

Introduction

The "Complete Generics" goal for Swift 3 has been fairly ill-defined thus far, with just this short blurb in the list of goals:

> *Complete generics: Generics are used pervasively in a number of Swift libraries, especially the standard library. However, there are a number of generics features the standard library requires to fully realize its vision, including recursive protocol constraints, the ability to make a constrained extension conform to a new protocol (i.e., an array of Equatable elements is Equatable), and so on. Swift 3.0 should provide those generics features needed by the standard library, because they affect the standard library's ABI.*

This message expands upon the notion of "completing generics". It is not a plan for Swift 3, nor an official core team communication, but it collects the results of numerous discussions among the core team and Swift developers, both of the compiler and the standard library. I hope to achieve several things:

- **Communicate a vision for Swift generics**, building on the original generics design document, so we have something concrete and comprehensive to discuss.
- **Establish some terminology** that the Swift developers have been using for these features, so our discussions can be more productive ("oh, you're proposing what we refer to as 'conditional conformances'; go look over at this thread").
- **Engage more of the community in discussions** of specific generics features, so we can coalesce around designs for public review. And maybe even get some of them implemented.

A message like this can easily turn into a centithread. To separate concerns in our discussion, I ask that replies to this specific thread be limited to discussions of the vision as a whole: how the pieces fit together, what pieces are missing, whether this is the right long-term vision for Swift, and so on. For discussions of specific language features, e.g., to work out the syntax and semantics of conditional conformances or discuss the implementation in compiler or use in the standard library, please start a new thread based on the feature names I'm using.

This message covers a lot of ground; I've attempted a rough categorization of the various features, and kept the descriptions brief to limit the overall length. Most of these aren't my ideas, and any syntax I'm providing is simply a way to express these ideas in code and is subject to change. Not all of these features will happen, either soon or ever, but they are intended to be a fairly complete whole that should mesh together. I've put a * next to features that I think are important in the nearer term vs. being interesting "some day". Mostly, the *'s reflect features that will have a significant impact on the Swift standard library's design and implementation.

Enough with the disclaimers; it's time to talk features.

Removing unnecessary restrictions

There are a number of restrictions to the use of generics that fall out of the implementation in the Swift compiler. Removal of these restrictions is a matter of implementation only; one need not introduce new syntax or semantics to realize them. I'm listing them for two reasons: first, it's an acknowledgment that these features are intended to exist in the model we have today, and, second, we'd love help with the implementation of these features.

Recursive protocol constraints (*)

This feature has been accepted in SE-0157 and is tracked by SR-1445.

Currently, an associated type cannot be required to conform to its enclosing protocol (or any protocol that inherits that protocol). For example, in the standard library `SubSequence` type of a `Sequence` should itself be a `Sequence`:

```
protocol Sequence {
  associatedtype Iterator : IteratorProtocol
  ...
  associatedtype SubSequence : Sequence   // currently ill-formed, but should be
possible
}
```

The compiler currently rejects this protocol, which is unfortunate: it effectively pushes the `SubSequence`-must-be-a-`Sequence` requirement into every consumer of `SubSequence`, and does not communicate the intent of this abstraction well.

Nested generics

This feature was tracked by SR-1446 and was released with Swift 3.1.

Currently, a generic type cannot be nested within another generic type, e.g.

```
struct X<T> {
  struct Y<U> { }
}
```

There isn't much to say about this: the compiler simply needs to be improved to handle nested generics throughout.

Concrete same-type requirements

This feature was tracked by SR-1009 and was released with Swift 3.1.

Currently, a constrained extension cannot use a same-type constraint to make a type parameter equivalent to a concrete type. For example:

```
extension Array where Element == String {
  func makeSentence() -> String {
    // uppercase first string, concatenate with spaces, add a period, whatever
  }
}
```

This is a highly-requested feature that fits into the existing syntax and semantics. Note that one could imagine introducing new syntax, e.g., extending `Array<String>`, which gets into new-feature territory: see the section on "Parameterized extensions".

Parameterizing other declarations

There are a number of Swift declarations that currently cannot have generic parameters; some of those have fairly natural extensions to generic forms that maintain their current syntax and semantics, but become more powerful when made generic.

Generic typealiases

This feature has been accepted in SE-0048 and was released with Swift 3.

Typealiases could be allowed to carry generic parameters. They would still be aliases (i.e., they would not introduce new types). For example:

```
typealias StringDictionary<Value> = Dictionary<String, Value>

var d1 = StringDictionary<Int>()
var d2: Dictionary<String, Int> = d1 // okay: d1 and d2 have the same type,
Dictionary<String, Int>
```

Generic associatedtypes

Associatedtypes could be allowed to carry generic parameters.

```
    protocol Wrapper {
      associatedtype Wrapped<T>

      static func wrap<T>(_ t: T) -> Wrapped<T>
    }
```

Generic associatedtypes would support all constraints supported by the language including where clauses. As with non-generic associatedtypes conforming types would be required to provide a nested type or typealias matching the name of the associatedtype. However, in this case the nested type or typealias would be generic.

```
    enum OptionalWrapper {
      typealias Wrapped<T> = Optional<T>

      static func wrap<T>(_ t: T) -> Optional<T>
    }
```

Note: generic associatedtypes address many use cases also addressed by higher-kinded types but with lower implementation complexity.

Generic subscripts

This feature has been accepted in SE-0148, was tracked by SR-115 and was released with Swift 4.

Subscripts could be allowed to have generic parameters. For example, we could introduce a generic subscript on a `Collection` that allows us to pull out the values at an arbitrary set of indices:

```
extension Collection {
  subscript<Indices: Sequence where Indices.Iterator.Element == Index>(indices:
Indices) -> [Iterator.Element] {
    get {
      var result = [Iterator.Element]()
      for index in indices {
        result.append(self[index])
      }
```

```
        return result
      }

      set {
        for (index, value) in zip(indices, newValue) {
          self[index] = value
        }
      }
    }
  }
```

Generic constants

`let` constants could be allowed to have generic parameters, such that they produce differently-typed values depending on how they are used. For example, this is particularly useful for named literal values, e.g.,

```
let π<T : ExpressibleByFloatLiteral>: T =
3.141592653589793238462643383279502884197169399
```

The Clang importer could make particularly good use of this when importing macros.

Parameterized extensions

Extensions themselves could be parameterized, which would allow some structural pattern matching on types. For example, this would permit one to extend an array of optional values, e.g.,

```
extension<T> Array where Element == T? {
  var someValues: [T] {
    var result = [T]()
    for opt in self {
      if let value = opt { result.append(value) }
    }
   return result
  }
}
```

We can generalize this to protocol extensions:

```
extension<T> Sequence where Element == T? {
  var someValues: [T] {
    var result = [T]()
    for opt in self {
      if let value = opt { result.append(value) }
    }
   return result
  }
}
```

Note that when one is extending nominal types, we could simplify the syntax somewhat to make the same-type constraint implicit in the syntax:

```
extension<T> Array<T?> {
  var someValues: [T] {
    var result = [T]()
    for opt in self {
      if let value = opt { result.append(value) }
    }
   return result
  }
}
```

When we're working with concrete types, we can use that syntax to improve the extension of concrete versions of generic types (per "Concrete same-type requirements", above), e.g.,

```
extension Array<String> {
  func makeSentence() -> String {
    // uppercase first string, concatenate with spaces, add a period, whatever
  }
}
```

Minor extensions

There are a number of minor extensions we can make to the generics system that don't fundamentally change what one can express in Swift, but which can improve its expressivity.

Arbitrary requirements in protocols (*)

This feature has been accepted in SE-0142 and was released with Swift 4.

Currently, a new protocol can inherit from other protocols, introduce new associated types, and add new conformance constraints to associated types (by redeclaring an associated type from an inherited protocol). However, one cannot express more general constraints. Building on the example from "Recursive protocol constraints", we really want the element type of a `Sequence`'s `SubSequence` to be the same as the element type of the `Sequence`, e.g.,

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

Hanging the `where` clause off the associated type protocol is not ideal, but that's a discussion for another thread.

Typealiases in protocols and protocol extensions (*)

This feature has been accepted in SE-0092 and was released with Swift 3.

Now that associated types have their own keyword (thanks!), it's reasonable to bring back `typealias`. Again with the `Sequence` protocol:

```
protocol Sequence {
  associatedtype Iterator : IteratorProtocol
  typealias Element = Iterator.Element   // rejoice! now we can refer to
SomeSequence.Element rather than SomeSequence.Iterator.Element
}
```

Default generic arguments

Generic parameters could be given the ability to provide default arguments, which would be used in cases where the type argument is not specified and type inference could not determine the type argument. For example:

```
public final class Promise<Value, Reason=Error> { ... }

func getRandomPromise() -> Promise<Int, Error> { ... }

var p1: Promise<Int> = ...
var p2: Promise<Int, Error> = p1     // okay: p1 and p2 have the same type
Promise<Int, Error>
var p3: Promise = getRandomPromise() // p3 has type Promise<Int, Error> due to type
inference
```

Generalized `class` constraints

This feature is a consequence of proposal SE-0156 and was released with Swift 4.

The `class` constraint can currently only be used for defining protocols. We could generalize it to associated type and type parameter declarations, e.g.,

```
protocol P {
  associatedtype A : class
}

func foo<T : class>(t: T) { }
```

As part of this, the magical `AnyObject` protocol could be replaced with an existential with a class bound, so that it becomes a typealias:

```
typealias AnyObject = protocol<class>
```

See the "Existentials" section, particularly "Generalized existentials", for more information.

Generalized supertype constraints

Currently, supertype constraints may only be specified using a concrete class or protocol type. This prevents us from abstracting over the supertype.

```
protocol P {
  associatedtype Base
  associatedtype Derived: Base
}
```

In the above example `Base` may be any type. `Derived` may be the same as `Base` or may be *any* subtype of `Base`. All subtype relationships supported by Swift should be supported in this context including (but not limited to) classes and subclasses, existentials and conforming concrete types or refining existentials, `T?` and `T`, `((Base) -> Void)` and `((Derived) -> Void)`, etc.

Generalized supertype constraints would be accepted in all syntactic locations where generic constraints are accepted.

Allowing subclasses to override requirements satisfied by defaults (*)

When a superclass conforms to a protocol and has one of the protocol's requirements satisfied by a member of a protocol extension, that member currently cannot be overridden by a subclass. For example:

```
protocol P {
  func foo()
}

extension P {
  func foo() { print("P") }
}

class C : P {
  // gets the protocol extension's
}

class D : C {
  /*override not allowed!*/ func foo() { print("D") }
}

let p: P = D()
p.foo() // gotcha: prints "P" rather than "D"!
```

`D.foo` should be required to specify "override" and should be called dynamically.

Major extensions to the generics model

Unlike the minor extensions, major extensions to the generics model provide more expressivity in the Swift generics system and, generally, have a much more significant design and implementation cost.

Conditional conformances (*)

This feature has been accepted in SE-0143 and is implemented in Swift 4.2.

Conditional conformances express the notion that a generic type will conform to a particular protocol only under certain circumstances. For example, `Array` is `Equatable` only when its elements are `Equatable`:

```
extension Array : Equatable where Element : Equatable { }
```

```
func ==<T : Equatable>(lhs: Array<T>, rhs: Array<T>) -> Bool { ... }
```

Conditional conformances are a potentially very powerful feature. One important aspect of this feature is how to deal with or avoid overlapping conformances. For example, imagine an adaptor over a `Sequence` that has conditional conformances to `Collection` and `MutableCollection`:

```
struct SequenceAdaptor<S: Sequence> : Sequence { }
extension SequenceAdaptor : Collection where S: Collection { ... }
extension SequenceAdaptor : MutableCollection where S: MutableCollection { }
```

This should almost certainly be permitted, but we need to cope with or reject "overlapping" conformances:

```
extension SequenceAdaptor : Collection where S: SomeOtherProtocolSimilarToCollection
{ } // trouble: two ways for SequenceAdaptor to conform to Collection
```

See the section on "Private conformances" for more about the issues with having the same type conform to the same protocol multiple times.

Variadic generics

Currently, a generic parameter list contains a fixed number of generic parameters. If one has a type that could generalize to any number of generic parameters, the only real way to deal with it today involves creating a set of types. For example, consider the standard library's `zip` function. It returns one of these when provided with two arguments to zip together:

```
public struct Zip2Sequence<Sequence1 : Sequence,
                           Sequence2 : Sequence> : Sequence { ... }

public func zip<Sequence1 : Sequence, Sequence2 : Sequence>(
              sequence1: Sequence1, _ sequence2: Sequence2)
            -> Zip2Sequence<Sequence1, Sequence2> { ... }
```

Supporting three arguments would require copy-pasting code:

```
public struct Zip3Sequence<Sequence1 : Sequence,
                           Sequence2 : Sequence,
                           Sequence3 : Sequence> : Sequence { ... }

public func zip<Sequence1 : Sequence, Sequence2 : Sequence, Sequence3 : Sequence>(
              sequence1: Sequence1, _ sequence2: Sequence2, _ sequence3: sequence3)
            -> Zip3Sequence<Sequence1, Sequence2, Sequence3> { ... }
```

Variadic generics would allow us to abstract over a set of generic parameters. The syntax below is hopelessly influenced by C++11 variadic templates (sorry), where putting an ellipsis ("...") to the left of a declaration makes it a "parameter pack" containing zero or more parameters and putting an ellipsis to the right of a type/expression/etc. expands the parameter packs within that type/expression into separate arguments. The important part is that we be able to meaningfully abstract over zero or more generic parameters, e.g.:

```
public struct ZipIterator<... Iterators : IteratorProtocol> : Iterator {  // zero or
more type parameters, each of which conforms to IteratorProtocol
  public typealias Element = (Iterators.Element...)                       // a tuple
containing the element types of each iterator in Iterators

  var (...iterators): (Iterators...)    // zero or more stored properties, one for
each type in Iterators
  var reachedEnd = false

  public mutating func next() -> Element? {
    if reachedEnd { return nil }

    guard let values = (iterators.next()...) else {   // call "next" on each of the
iterators, put the results into a tuple named "values"
      reachedEnd = true
      return nil
    }

    return values
  }
}

public struct ZipSequence<...Sequences : Sequence> : Sequence {
  public typealias Iterator = ZipIterator<Sequences.Iterator...>   // get the zip
iterator with the iterator types of our Sequences

  var (...sequences): (Sequences...)    // zero or more stored properties, one for
each type in Sequences

  // details ...
}
```

Such a design could also work for function parameters, so we can pack together multiple function arguments with different types, e.g.,

```
public func zip<... Sequences : SequenceType>(... sequences: Sequences...)
            -> ZipSequence<Sequences...> {
  return ZipSequence(sequences...)
}
```

Finally, this could tie into the discussions about a tuple "splat" operator. For example:

```
func apply<... Args, Result>(fn: (Args...) -> Result,    // function taking some
number of arguments and producing Result
                           args: (Args...)) -> Result {  // tuple of arguments
  return fn(args...)                                     // expand the arguments in
the tuple "args" into separate arguments
}
```

Extensions of structural types

Currently, only nominal types (classes, structs, enums, protocols) can be extended. One could imagine extending structural types--particularly tuple types--to allow them to, e.g., conform to protocols. For example, pulling together variadic generics, parameterized extensions, and conditional conformances, one could express "a tuple type is `Equatable` if all of its element types are `Equatable`":

```
extension<...Elements : Equatable> (Elements...) : Equatable {   // extending the
tuple type "(Elements...)" to be Equatable
}
```

There are some natural bounds here: one would need to have actual structural types. One would not be able to extend every type:

```
extension<T> T { // error: neither a structural nor a nominal type
}
```

And before you think you're cleverly making it possible to have a conditional conformance that makes every type `T` that conforms to protocol `P` also conform to protocol `Q`, see the section "Conditional conformances via protocol extensions", below:

```
extension<T : P> T : Q { // error: neither a structural nor a nominal type
}
```

Syntactic improvements

There are a number of potential improvements we could make to the generics syntax. Such a list could go on for a very long time, so I'll only highlight some obvious ones that have been discussed by the Swift developers.

Default implementations in protocols (*)

Currently, protocol members can never have implementations. We could allow one to provide such implementations to be used as the default if a conforming type does not supply an implementation, e.g.,

```
protocol Bag {
  associatedtype Element : Equatable
  func contains(element: Element) -> Bool

  func containsAll<S: Sequence where Sequence.Iterator.Element == Element>(elements:
S) -> Bool {
    for x in elements {
      if contains(x) { return true }
    }
    return false
  }
}

struct IntBag : Bag {
  typealias Element = Int
  func contains(element: Int) -> Bool { ... }

  // okay: containsAll requirement is satisfied by Bag's default implementation
}
```

One can get this effect with protocol extensions today, hence the classification of this feature as a (mostly) syntactic improvement:

```
protocol Bag {
  associatedtype Element : Equatable
  func contains(element: Element) -> Bool

  func containsAll<S: Sequence where Sequence.Iterator.Element == Element>(elements:
S) -> Bool
}

extension Bag {
  func containsAll<S: Sequence where Sequence.Iterator.Element == Element>(elements:
S) -> Bool {
    for x in elements {
      if contains(x) { return true }
    }
    return false
  }
}
```

Moving the `where` clause outside of the angle brackets (*)

Accepted in SE-0081 and implemented in Swift 3.

The `where` clause of generic functions comes very early in the declaration, although it is generally of much less concern to the client than the function parameters and result type that follow it. This is one of the things that contributes to "angle bracket blindness". For example, consider the `containsAll` signature above:

```
func containsAll<S: Sequence where Sequence.Iterator.Element == Element>(elements:
S) -> Bool
```

One could move the `where` clause to the end of the signature, so that the most important parts--name, generic parameter, parameters, result type--precede it:

```
func containsAll<S: Sequence>(elements: S) -> Bool
       where Sequence.Iterator.Element == Element
```

Renaming `protocol<...>` to `Any<...>` (*)

Accepted in SE-0095 as "Replace `protocol<P1,P2>` syntax with `P1 & P2` syntax" and implemented in Swift 3.

The `protocol<...>` syntax is a bit of an oddity in Swift. It is used to compose protocols together, mostly to create values of existential type, e.g.,

```
var x: protocol<NSCoding, NSCopying>
```

It's weird that it's a type name that starts with a lowercase letter, and most Swift developers probably never deal with this feature unless they happen to look at the definition of `Any`:

```
typealias Any = protocol<>
```

"Any" might be a better name for this functionality. `Any` without brackets could be a keyword for "any type", and "Any" followed by brackets could take the role of `protocol<>` today:

```
var x: Any<NSCoding, NSCopying>
```

That reads much better: "Any type that conforms to `NSCoding` and `NSCopying`". See the section "Generalized existentials" for additional features in this space.

Maybe

There are a number of features that get discussed from time-to-time, while they could fit into Swift's generics system, it's not clear that they belong in Swift at all. The important question for any feature in this category is not "can it be done" or "are there cool things we can express", but "how can everyday Swift developers benefit from the addition of such a feature?". Without strong motivating examples, none of these "maybes" will move further along.

Dynamic dispatch for members of protocol extensions

Only the requirements of protocols currently use dynamic dispatch, which can lead to surprises:

```
protocol P {
  func foo()
}

extension P {
  func foo() { print("P.foo()") }
  func bar() { print("P.bar()") }
}

struct X : P {
  func foo() { print("X.foo()") }
  func bar() { print("X.bar()") }
}

let x = X()
x.foo() // X.foo()
x.bar() // X.bar()

let p: P = X()
p.foo() // X.foo()
p.bar() // P.bar()
```

Swift could adopt a model where members of protocol extensions are dynamically dispatched.

Named generic parameters

When specifying generic arguments for a generic type, the arguments are always positional: `Dictionary<String, Int>` is a `Dictionary` whose `Key` type is `String` and whose `Value` type is `Int`, by convention. One could permit the arguments to be labeled, e.g.,

```
var d: Dictionary<Key: String, Value: Int>
```

Such a feature makes more sense if Swift gains default generic arguments, because generic argument labels would allow one to skip defaulted arguments.

Generic value parameters

Currently, Swift's generic parameters are always types. One could imagine allowing generic parameters that are values, e.g.,

```
struct MultiArray<T, let Dimensions: Int> { // specify the number of dimensions to
the array
  subscript (indices: Int...) -> T {
    get {
      require(indices.count == Dimensions)
      // ...
    }
}
```

A suitably general feature might allow us to express fixed-length array or vector types as a standard library component, and perhaps also allow one to implement a useful dimensional analysis library. Tackling this feature potentially means determining what it is for an expression to be a "constant expression" and diving into dependent-typing, hence the "maybe".

Higher-kinded types

Higher-kinded types allow one to express the relationship between two different specializations of the same nominal type within a protocol. For example, if we think of the `Self` type in a protocol as really being `Self<T>`, it allows us to talk about the relationship between `Self<T>` and `Self<U>` for some other type `U`. For example, it could allow the `map` operation on a collection to return a collection of the same kind but with a different operation, e.g.,

```
let intArray: Array<Int> = ...
intArray.map { String($0) } // produces Array<String>
let intSet: Set<Int> = ...
intSet.map { String($0) }   // produces Set<String>
```

Potential syntax borrowed from one thread on higher-kinded types uses `~=` as a "similarity" constraint to describe a `Functor` protocol:

```
protocol Functor {
  associatedtype A
  func fmap<FB where FB ~= Self>(f: A -> FB.A) -> FB
}
```

Specifying type arguments for uses of generic functions

Not in scope for Swift 4.

The type arguments of a generic function are always determined via type inference. For example, given:

```
func f<T>(t: T)
```

one cannot directly specify `T`: either one calls `f` (and `T` is determined via the argument's type) or one uses `f` in a context where it is given a particular function type (e.g., `let x: (Int) -> Void = f` would infer `T = Int`). We could permit explicit specialization here, e.g.,

```
let x = f<Int> // x has type (Int) -> Void
```

Unlikely

Features in this category have been requested at various times, but they don't fit well with Swift's generics system because they cause some part of the model to become overly complicated, have unacceptable implementation limitations, or overlap significantly with existing features.

Generic protocols

One of the most commonly requested features is the ability to parameterize protocols themselves. For example, a protocol that indicates that the `Self` type can be constructed from some specified type `T`:

```
protocol ConstructibleFromValue<T> {
  init(_ value: T)
}
```

Implicit in this feature is the ability for a given type to conform to the protocol in two different ways. A `Real` type might be constructible from both `Float` and `Double`, e.g.,

```
struct Real { ... }
extension Real : ConstructibleFrom<Float> {
  init(_ value: Float) { ... }
}
extension Real : ConstructibleFrom<Double> {
  init(_ value: Double) { ... }
}
```

Most of the requests for this feature actually want a different feature. They tend to use a parameterized `Sequence` as an example, e.g.,

```
protocol Sequence<Element> { ... }

func foo(strings: Sequence<String>) {  /// works on any sequence containing Strings
  // ...
}
```

The actual requested feature here is the ability to say "Any type that conforms to `Sequence` whose `Element` type is `String`", which is covered by the section on "Generalized existentials", below.

More importantly, modeling `Sequence` with generic parameters rather than associated types is tantalizing but wrong: you don't want a type conforming to `Sequence` in multiple ways, or (among other things) your `for..in` loops stop working, and you lose the ability to dynamically cast down to an existential `Sequence` without binding the `Element` type (again, see "Generalized existentials"). Use cases similar to the `ConstructibleFromValue` protocol above seem too few to justify the potential for confusion between associated types and generic parameters of protocols; we're better off not having the latter.

Private conformances

Right now, a protocol conformance can be no less visible than the minimum of the conforming type's access and the protocol's access. Therefore, a public type conforming to a public protocol must provide the conformance publicly. One could imagine removing that restriction, so that one could introduce a private conformance:

```
public protocol P { }
public struct X { }
extension X : internal P { ... } // X conforms to P, but only within this module
```

The main problem with private conformances is the interaction with dynamic casting. If I have this code:

```
func foo(value: Any) {
  if let x = value as? P { print("P") }
}

foo(X())
```

Under what circumstances should it print "P"? If `foo()` is defined within the same module as the conformance of `X` to `P`? If the call is defined within the same module as the conformance of `X` to `P`? Never? Either of the first two answers requires significant complications in the dynamic casting infrastructure to take into account the module in which a particular dynamic cast occurred (the first option) or where an existential was formed (the second option), while the third answer breaks the link between the static and dynamic type systems--none of which is an acceptable result.

Retroactive protocol refinement

We often get requests to make protocols retroactively refine other protocols. For example:

```
protocol P {
  func foo()
}

protocol Q {
  func bar()
}

extension Q : P { // Make every type that conforms to Q also conforms to P
  func foo() {    // Implement `P.foo` requirement in terms of `Q.bar`
    bar()
  }
}

func f<T: P>(t: T) { ... }

struct X : Q {
  func bar() { ... }
}

f(X()) // okay: X conforms to P through the conformance of Q to P
```

This is an extremely powerful feature: it allows one to map the abstractions of one domain into another domain (e.g., every `Matrix` is a `Graph`). However, similar to private conformances, it puts a major burden on the dynamic-casting runtime to chase down arbitrarily long and potentially cyclic chains of conformances, which makes efficient implementation nearly impossible.

Potential removals

The generics system doesn't seem like a good candidate for a reduction in scope; most of its features do get used fairly pervasively in the standard library, and few feel overly anachronistic. However...

Associated type inference

SE-0108, a proposal to remove this feature, was rejected.

Associated type inference is the process by which we infer the type bindings for associated types from other requirements. For example:

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

struct IntIterator : IteratorProtocol {
  mutating func next() -> Int? { ... }  // use this to infer Element = Int
}
```

Associated type inference is a useful feature. It's used throughout the standard library, and it helps keep associated types less visible to types that simply want to conform to a protocol. On the other hand, associated type inference is the only place in Swift where we have a global type inference problem: it has historically been a major source of bugs, and implementing it fully and correctly requires a drastically different architecture to the type checker. Is the value of this feature worth keeping global type inference in the Swift language, when we have deliberatively avoided global type inference elsewhere in the language?

Existentials

Existentials aren't really generics per se, but the two systems are closely intertwined due to their mutual dependence on protocols.

Generalized existentials

The restrictions on existential types came from an implementation limitation, but it is reasonable to allow a value of protocol type even when the protocol has Self constraints or associated types. For example, consider `IteratorProtocol` again and how it could be used as an existential:

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

let it: IteratorProtocol = ...
it.next()   // if this is permitted, it could return an "Any?", i.e., the
existential that wraps the actual element
```

Additionally, it is reasonable to want to constrain the associated types of an existential, e.g., "a `Sequence` whose element type is `String`" could be expressed by putting a where clause into `protocol<...>` or `Any<...>` (per "Renaming `protocol<...>` to `Any<...>`"):

```
let strings: Any<Sequence where .Iterator.Element == String> = ["a", "b", "c"]
```

The leading `.` indicates that we're talking about the dynamic type, i.e., the `Self` type that's conforming to the `Sequence` protocol. There's no reason why we cannot support arbitrary `where` clauses within the `Any<...>`. This very-general syntax is a bit unwieldy, but common cases can easily be wrapped up in a generic typealias (see the section "Generic typealiases" above):

```
typealias AnySequence<Element> = Any<Sequence where .Iterator.Element == Element>
let strings: AnySequence<String> = ["a", "b", "c"]
```

Opening existentials

Generalized existentials as described above will still have trouble with protocol requirements that involve `Self` or associated types in function parameters. For example, let's try to use `Equatable` as an existential:

```
protocol Equatable {
  func ==(lhs: Self, rhs: Self) -> Bool
  func !=(lhs: Self, rhs: Self) -> Bool
}

let e1: Equatable = ...
let e2: Equatable = ...
if e1 == e2 { ... } // error: e1 and e2 don't necessarily have the same dynamic type
```

One explicit way to allow such operations in a type-safe manner is to introduce an "open existential" operation of some sort, which extracts and gives a name to the dynamic type stored inside an existential. For example:

```
if let storedInE1 = e1 openas T { // T is the type of storedInE1, a copy of the
value stored in e1
  if let storedInE2 = e2 as? T {  // Does e2 have type T? If so, copy its value to
storedInE2
```

```
    if storedInE1 == storedInE2 { ... } // Okay: storedInT1 and storedInE2 are both
of type T, which we know is Equatable
  }
}
```