Generics in Swift

Motivation

Most types and functions in code are expressed in terms of a single, concrete set of sets. Generics generalize this notion by allowing one to express types and functions in terms of an abstraction over a (typically unbounded) set of types, allowing improved code reuse. A typical example of a generic type is a linked list of values, which can be used with any type of value. In C++, this might be expressed as:

```
template<typename T>
class List {
public:
  struct Node {
    T value;
    Node *next;
  };

  Node *first;
};
```

where List<Int>, List<String>, and List<DataRecord> are all distinct types that provide a linked list storing integers, strings, and DataRecords, respectively. Given such a data structure, one also needs to be able to implement generic functions that can operate on a list of any kind of elements, such as a simple, linear search algorithm:

```
template<typename T>
typename List<T>::Node *find(const List<T>&list, const T& value) {
  for (typename List<T>::Node *result = list.first; result; result = result->next)
    if (result->value == value)
      return result;

  return 0;
}
```

Generics are important for the construction of useful libraries, because they allow the library to adapt to application-specific data types without losing type safety. This is especially important for foundational libraries containing common data structures and algorithms, since these libraries are used across nearly every interesting application.

The alternatives to generics tend to lead to poor solutions:

- Object-oriented languages tend to use "top" types (id in Objective-C, java.lang.Object in pre-generics Java, etc.) for their containers and algorithms, which gives up static type safety. Pre-generics Java forced the user to introduce run-time-checked type casts when interacting with containers (which is overly verbose), while Objective-C relies on id's unsound implicit conversion behavior to eliminate the need for casts.
- Many languages bake common data structures (arrays, dictionaries, tables) into the language itself. This is unfortunate both because it significantly increases the size of the core language and because users then tend to use this limited set of data structures for *every* problem, even when another (not-baked-in) data structure would be better.

Swift is intended to be a small, expressive language with great support for building libraries. We'll need generics to be able to build those libraries well.

Goals

- Generics should enable the development of rich generic libraries that feel similar to first-class language features
- Generics should work on any type, whether it is a value type or some kind of object type
- Generic code should be almost as easy to write as non-generic code
- Generic code should be compiled such that it can be executed with any data type without requiring a separate "instantiation" step
- Generics should interoperate cleanly with run-time polymorphism
- Types should be able to retroactively modified to meet the requirements of a generic algorithm or data structure

As important as the goals of a feature are the explicit non-goals, which we don't want or don't need to support:

- Compile-time "metaprogramming" in any form
- Expression-template tricks a la Boost.Spirit, POOMA

Polymorphism

Polymorphism allows one to use different data types with a uniform interface. Overloading already allows a form of polymorphism (ad hoc polymorphism) in Swift. For example, given:

```
func +(x : Int, y : Int) -> Int { add... }
func +(x : String, y : String) -> String { concat... }
```

we can write the expression "x + y", which will work for both integers and strings.

However, we want the ability to express an algorithm or data structure independently of mentioning any data type. To do so, we need a way to express the essential interface that algorithm or data structure requires. For example, an accumulation algorithm would need to express that for any type T, one can write the expression "x + y" (where x and y are both of type T) and it will produce another T.

Protocols

Most languages that provide some form of polymorphism also have a way to describe abstract interfaces that cover a range of types: Java and C# interfaces, C++ abstract base classes, Objective-C protocols, Scala traits, Haskell type classes, C++ concepts (briefly), and many more. All allow one to describe functions or methods that are part of the interface, and provide some way to re-use or extend a previous interface by adding to it. We'll start with that core feature, and build onto it what we need.

In Swift, I suggest that we use the term protocol for this feature, because I expect the end result to be similar enough to Objective-C protocols that our users will benefit, and (more importantly) different enough from Java/C# interfaces and C++ abstract base classes that those terms will be harmful. The term trait comes with the wrong connotation for C++ programmers, and none of our users know Scala.

In its most basic form, a protocol is a collection of function signatures:

```
protocol Document {
  func title() -> String
}
```

Document describes types that have a title() operation that accepts no arguments and returns a String. Note that there is implicitly a 'self' type, which is the type that conforms to the protocol itself. This follows how most object-oriented languages describe interfaces, but deviates from Haskell type classes and C++ concepts, which require explicit type parameters for all of the types. We'll revisit this decision later.

Protocol Inheritance

Composition of protocols is important to help programmers organize and understand a large number of protocols and the data types that conform to those protocols. For example, we could extend our Document protocol to cover documents that support versioning:

```
protocol VersionedDocument : Document {
  func version() -> Int
}
```

Multiple inheritance is permitted, allowing us to form a directed acyclic graph of protocols:

```
protocol PersistentDocument : VersionedDocument, Serializable {
  func saveToFile(_ filename : path)
}
```

Any type that conforms to PersistentDocument also conforms to VersionedDocument, Document, and Serializable, which gives us substitutability.

Self Types

Protocols thus far do not give us an easy way to express simple binary operations. For example, let's try to write a Comparable protocol that could be used to search for a generic find() operation:

```
protocol Comparable {
  func isEqual(_ other : ???) -> Bool
}
```

Our options for filling in ??? are currently very poor. We could use the syntax for saying "any type" or "any type that is comparable", as one must do most OO languages, including Java, C#, and Objective-C, but that's not expressing what we want: that the type of both of the arguments be the same. This is sometimes referred to as the binary method problem (http://www.cis.upenn.edu/~bcpierce/papers/binary.ps has a discussion of this problem, including the solution I'm proposing below).

Neither C++ concepts nor Haskell type classes have this particular problem, because they don't have the notion of an implicit 'Self' type. Rather, they explicitly parameterize everything. In C++ concepts:

```
concept Comparable<typename T> {
  bool T::isEqual(T);
}
```

Java and C# programmers work around this issue by parameterizing the interface, e.g. (in Java):

```
abstract class Comparable<THIS extends Comparable<THIS>> {
  public bool isEqual(THIS other);
}
```

and then a class X that wants to be Comparable will inherit from Comparable<X>. This is ugly and has a number of pitfalls; see http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6479372.

Scala and Strongtalk have the notion of the 'Self' type, which effectively allows one to refer to the eventual type of 'self' (which we call 'self'). 'Self' (which we call 'Self' in Swift) allows us to express the Comparable protocol in a natural way:

```
protocol Comparable {
  func isEqual(_ other : Self) -> Bool
}
```

By expressing Comparable in this way, we know that if we have two objects of type T where T conforms to Comparable, comparison between those two objects with isEqual is well-typed. However, if we have objects of different types T and U, we cannot compare those objects with isEqual even if both T and U are Comparable.

Self types are not without their costs, particularly in the case where Self is used as a parameter type of a class method that will be subclassed. Here, the parameter type ends up being (implicitly) covariant, which tightens up type-checking but may also force us into more dynamic type checks. We can explore this separately; within protocols, type-checking for Self is more direct.

Associated Types

In addition to Self, a protocol's operations often need to refer to types that are related to the type of 'Self', such as a type of data stored in a collection, or the node and edge types of a graph. For example, this would allow us to cleanly describe a protocol for collections:

```
protocol Collection {
  typealias Element
  func forEach(_ callback : (value : Element) -> Void)
  func add(_ value : Element)
}
```

It is important here that a generic function that refers to a given type T, which is known to be a collection, can access the associated types corresponding to T. For example, one could implement an "accumulate" operation for an arbitrary Collection, but doing so requires us to specify some constraints on the Value type of the collection. We'll return to this later.

Operators, Properties, and Subscripting

As previously noted, protocols can contain both function requirements (which are in effect requirements for instance methods) and associated type requirements. Protocols can also contain operators, properties, and subscript operators:

```
protocol RandomAccessContainer : Collection {
  var count: Int
  func == (lhs: Self, rhs: Self)
  subscript(i: Int) -> Element
}
```

Operator requirements can be satisfied by operator definitions, property requirements can be satisfied by either variables or properties, and subscript requirements can be satisfied by subscript operators.

Conforming to a Protocol

Thus far, we have not actually shown how a type can meet the requirements of a protocol. The most syntactically lightweight approach is to allow implicit conformance. This is essentially duck typing, where a type is assumed to conform to a protocol if it meets the syntactic requirements of the protocol. For example, given:

```
protocol Shape {
  func draw()
}
```

One could write a Circle struct such as:

```
struct Circle {
  var center : Point
  var radius : Int

  func draw() {
    // draw it
  }
}
```

Circle provides a draw() method with the same input and result types as required by the Shape protocol. Therefore, Circle conforms to Shape.

Implicit protocol conformance is convenient, because it requires no additional typing. However, it can run into some trouble when an entity that syntactically matches a protocol doesn't provide the required semantics. For example, Cowboys also know how to "draw!":

```
struct Cowboy {
  var gun : SixShooter

  func draw() {
    // draw!
  }
}
```

It is unlikely that Cowboy is meant to conform to Shape, but the method name and signatures match, so implicit conformance deduces that Cowboy conforms to Shape. Random collisions between types are fairly rare. However, when one is using protocol inheritance with fine-grained (semantic or mostly-semantic) differences between protocols in the hierarchy, they become more common. See http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1798.html for examples of this problem as it surfaced with C++ concepts. It is not clear at this time whether we want implicit conformance in Swift: there's no existing code to worry about, and explicit conformance (described below) provides some benefits.

Explicit Protocol Conformance

Type authors often implement types that are intended to conform to a particular protocol. For example, if we want a linked-list type to conform to Collection, we can specify that it is by adding a protocol conformance annotation to the type:

```
struct EmployeeList : Collection { // EmployeeList is a collection
  typealias Element = T
  func forEach(_ callback : (value : Element) -> Void) { /* Implement this */ }
  func add(_ value : Element) { /* Implement this */ }
}
```

This explicit protocol conformance declaration forces the compiler to check that EmployeeList actually does meet the requirements of the Collection protocol. If we were missing an operation (say, forEach) or had the wrong signature, the definition of 'EmployeeList' would be ill-formed. Therefore, explicit conformance provides both documentation for the user of EmployeeList and checking for the author and future maintainers of EmployeeList.

Any nominal type (such as an enum, struct, or class) can be specified to conform to one or more protocols in this manner. Additionally, a typealias can be specified to conform to one or more protocols, e.g.,:

```
typealias NSInteger : Numeric = Int
```

While not technically necessary due to retroactive modeling (below), this can be used to document and check that a particular type alias does in fact meet some basic, important requirements. Moreover, it falls out of the syntax that places requirements on associated types.

Retroactive Modeling

When using a set of libraries, it's fairly common that one library defines a protocol (and useful generic entities requiring that protocol) while another library provides a data type that provides similar functionality to that protocol, but under a different name. Retroactive modeling is the process by which the type is retrofitted (without changing the type) to meet the requirements of the protocol.

In Swift, we provide support for retroactive modeling by allowing extensions, e.g.,:

```
extension String : Collection {
  typealias Element = char
  func forEach(_ callback : (value : Element) -> Void) { /* use existing String routines to enumerate characte
  func add(_ value : Element) { self += value /* append character */ }
}
```

Once an extension is defined, the extension now conforms to the Collection protocol, and can be used anywhere a Collection is expected.

Default Implementations

The functions declared within a protocol are requirements that any type must meet if it wants to conform to the protocol. There is a natural tension here, then, between larger protocols that make it easier to write generic algorithms, and smaller protocols that make it easier to write conforming types. For example, should a Numeric protocol implement all operations, e.g.,:

```
protocol Numeric {
  func +(lhs : Self, rhs : Self) -> Self
  func -(lhs : Self, rhs : Self) -> Self
  func +(x : Self) -> Self
  func -(x : Self) -> Self
}
```

which would make it easy to write general numeric algorithms, but requires the author of some BigInt class to implement a lot of functionality, or should the numeric protocol implement just the core operations:

```
protocol Numeric {
  func +(lhs : Self, rhs : Self) -> Self
  func -(x : Self) -> Self
}
```

to make it easier to adopt the protocol (but harder to write numeric algorithms)? Both of the protocols express the same thing (semantically), because one can use the core operations (binary +, unary -) to implement the other algorithms. However, it's far easier to allow the protocol itself to provide default implementations:

```
protocol Numeric {
  func +(lhs : Self, rhs : Self) -> Self
  func -(lhs : Self, rhs : Self) -> Self { return lhs + -rhs }
  func +(x : Self) -> Self { return x }
  func -(x : Self) -> Self
}
```

This makes it easier both to implement generic algorithms (which can use the most natural syntax) and to make a new type conform to the protocol. For example, if we were to define only the core algorithms in our BigNum type:

```
struct BigNum : Numeric {
  func +(lhs : BigNum, rhs : BigNum) -> BigNum { ... }
  func -(x : BigNum) -> BigNum { ... }
}
```

the compiler will automatically synthesize the other operations needed for the protocol. Moreover, these operations will be available to uses of the BigNum class as if they had been written in the type itself (or in an extension of the type, if that feature is used), which means that protocol conformance actually makes it easier to define types that conform to protocols, rather than just providing additional checking.

Subtype Polymorphism

Subtype polymorphism is based on the notion of substitutability. If a type S is a subtype of a type T, then a value of type S can safely be used where a value of type T is expected. Object-oriented languages typically use subtype polymorphism, where the subtype relationship is based on inheritance: if the class Dog inherits from the class Animal, then Dog is a subtype of Animal. Subtype polymorphism is generally dynamic, in the sense that the substitution occurs at run-time, even if it is statically type-checked.

In Swift, we consider protocols to be types. A value of protocol type has an existential type, meaning that we don't know the concrete type until run-time (and even then it varies), but we know that the type conforms to the given protocol. Thus, a variable can be declared with type "Serializable", e.g.,:

```
var x : Serializable = // value of any Serializable type
x.serialize() // okay: serialize() is part of the Serializable protocol
```

Naturally, such polymorphism is dynamic, and will require boxing of value types to implement. We can now see how Self types interact with subtype polymorphism. For example, say we have two values of type Comparable, and we try to compare them:

```
var x : Comparable = ...
var y : Comparable = ...
if x.isEqual(y) { // well-typed?
}
```

Whether x.isEqual(y) is well-typed is not statically determinable, because the dynamic type of x may different from the dynamic type of y, even if they are both comparable (e.g., one is an Int and the other a String). It can be implemented by the compiler as a dynamic type check, with some general failure mode (aborting, throwing an exception, etc.) if the dynamic type check fails.

To express types that meet the requirements of several protocols, one can just create a new protocol aggregating those protocols:

```
protocol SerializableDocument : Document, Serializable { }
var doc : SerializableDocument
print(doc.title()) // okay: title() is part of the Document protocol, so we can call it
doc.serialize(stout) // okay: serialize() is part of the Serializable protocol
```

However, this only makes sense when the resulting protocol is a useful abstraction. A SerializableDocument may or may not be a useful abstraction. When it is not useful, one can instead use '&' types to compose different protocols, e.g.,:

```
var doc : Document & Serializable
```

Here, doc has an existential type that is known to conform to both the Document and Serializable protocols. This gives rise to a natural "top" type, such that every type in the language is a subtype of "top". Java has java.lang.Object, C# has object, Objective-C has "id" (although "id" is weird, because it is also convertible to everything; it's best not to use it as a model). In Swift, the "top" type is simply an empty protocol composition: `Any`:

```
var value : Any = 17 // an any can hold an integer
value = "hello" // or a String
value = (42, "hello", Red) // or anything else
```

Bounded Parametric Polymorphism

Parametric polymorphism is based on the idea of providing type parameters for a generic function or type. When using that function or type, one substitutes concrete types for the type parameters. Strictly speaking, parametric polymorphism allows *any* type to be substituted for a type parameter, but it's useless in practice because that means that generic functions or types cannot do anything to the type parameters: they must instead rely on first-class functions passed into the generic function or type to perform any meaningful work.

Far more useful (and prevalent) is bounded parametric polymorphism, which allows the generic function or type to specify constraints (bounds) on the type parameters. By specifying these bounds, it becomes far easier to write and use these generic functions and types. Haskell type classes, Java and C# generics, C++ concepts, and many other language features support bounded parametric polymorphism.

Protocols provide a natural way to express the constraints of a generic function in Swift. For example, one could define a generic linked list as:

```
struct ListNode<T> {
  var Value : T
  enum NextNode { case Node : ListNode<T>, End }
  var Next : NextNode
}

struct List<T > {
  var First : ListNode<T>::NextNode
}
```

This list works on any type T. One could then add a generic function that inserts at the beginning of the list:

```
func insertAtBeginning<T>(_ list : List<T>, value : T) {
  list.First = ListNode<T>(value, list.First)
}
```

Expressing Constraints

Within the type parameter list of a generic type or function (e.g., the <T> in ListNode<T>), the 'T' introduces a new type parameter and the (optional) ": type" names a protocol (or protocol composition) to which 'T' must conform. Within the body of the generic type or function, any of the functions or types described by the constraints are available. For example, let's implement a find() operation on lists:

```
func find<T : Comparable>(_ list : List<T>, value : T) -> Int {
  var index = 0
  var current
  for (current = list.First; current is Node; current = current.Next) {
    if current.Value.isEqual(value) { // okay: T is Comparable
      return index
    }
    index = index + 1
  }
  return -1
}
```

In addition to providing constraints on the type parameters, we also need to be able to constrain associated types. To do so, we introduce the notion of a "where" clause, which follows the signature of the generic type or function. For example, let's generalize our find algorithm to work on any ordered collection:

```
protocol OrderedCollection : Collection {
  func size() -> Int
  func getAt(_ index : Int) -> Element // Element is an associated type
}

func find<C : OrderedCollection where C.Element : Comparable>(
       _ collection : C, value : C.Element) -> Int
{
  for index in 0...collection.size() {
    if (collection.getAt(index) == value) { // okay: we know that C.Element is Comparable
      return index
    }
  }
  return -1
}
```

The where clause is actually the more general way of expressing constraints, and the constraints expressed in the angle brackets (e.g., <C : OrderedCollection>) are just sugar for a where clause. For example, the above find() signature is equivalent to:

```
func find<C where C : OrderedCollection, C.Element : Comparable>(
       _ collection : C, value : C.Element) -> Int
```

Note that find<C> is shorthand for (and equivalent to) find<C : Any>, since every type conforms to the Any protocol composition.

There are two other important kinds of constraints that need to be expressible. Before we get to those, consider a simple "Enumerator" protocol that lets us describe an iteration of values of some given value type:

```
protocol Enumerator {
  typealias Element
  func isEmpty() -> Bool
  func next() -> Element
}
```

Now, we want to express the notion of an enumerable collection, which provides iteration, which we do by adding requirements into the protocol:

```
protocol EnumerableCollection : Collection {
  typealias EnumeratorType : Enumerator
  where EnumeratorType.Element == Element
  func getEnumeratorType() -> EnumeratorType
}
```

Here, we are specifying constraints on an associated type (EnumeratorType must conform to the Enumerator protocol), by adding a conformance clause (: Enumerator) to the associated type definition. We also use a separate where clause to require that the type of values produced by querying the enumerator is the same as the type of values stored in the container. This is important, for example, for use with the Comparable protocol (and any protocol using Self types), because it maintains type identity within the generic function or type.

Constraint Inference

Generic types often constrain their type parameters. For example, a SortedDictionary, which provides dictionary functionality using some kind of balanced binary tree (as in C++'s std::map), would require that its key type be Comparable:

```
class SortedDictionary<Key : Comparable, Value> {
  // ...
}
```

Naturally, one any generic operation on a SortedDictionary<K,V> would also require that K be Comparable, e.g.,:

```
func forEachKey<Key : Comparable, Value>(_ c : SortedDictionary<Key, Value>,
                                         f : (Key) -> Void) { /* ... */ }
```

However, explicitly requiring that Key conform to Comparable is redundant: one could not provide an argument for 'c' without the Key type of the SortedDictionary conforming to Comparable, because the SortedDictionary type itself could not be formed. Constraint inference infers these additional constraints within a generic function from the parameter and return types of the function, simplifying the specification of forEachKey:

```
func forEachKey<Key, Value>(_ c : SortedDictionary<Key, Value>,
                            f : (Key) -> Void) { /* ... */ }
```

Type Parameter Deduction

As noted above, type arguments will be deduced from the call arguments to a generic function:

```
var values : list<Int>
insertAtBeginning(values, 17) // deduces T = Int
```

Since Swift already has top-down type inference (as well as the C++-like bottom-up inference), we can also deduce type arguments from the result type:

```
func cast<T, U>(_ value : T) -> U { ... }
var x : Any
var y : Int = cast(x) // deduces T = Any, U = Int
```

We require that all type parameters for a generic function be deducible. We introduce this restriction so that we can avoid introducing a syntax for explicitly specifying type arguments to a generic function, e.g.,:

```
var y : Int = cast<Int>(x) // not permitted: < is the less-than operator
```

This syntax is horribly ambiguous in C++, and with good type argument deduction, should not be necessary in Swift.

Implementation Model

Because generics are constrained, a well-typed generic function or type can be translated into object code that uses dynamic dispatch to perform each of its operations on type parameters. This is in stark contrast to the instantiation model of C++ templates, where each new set of template arguments requires the generic function or type to be compiled again. This model is important for scalability of builds, so that the time to perform type-checking and code generation scales with the amount of code written rather than the amount of code instantiated. Moreover, it can lead to smaller binaries and a more flexible language (generic functions can be "virtual").

The translation model is fairly simple. Consider the generic find() we implemented for lists, above:

```
func find<T : Comparable>(_ list : List<T>, value : T) -> Int {
  var index = 0
  var current = list.First
  while current is ListNode<T> { // now I'm just making stuff up
    if current.value.isEqual(value) { // okay: T is Comparable
      return index
    }
    current = current.Next
    index = index + 1
  }
  return -1
}
```

to translate this into executable code, we form a vtable for each of the constraints on the generic function. In this case, we'll have a vtable for Comparable T. Every operation within the body of this generic function type-checks to either an operation on some concrete type (e.g., the operations on Int), to an operation within a protocol (which requires indirection through the corresponding vtable), or to an operation on a generic type definition, all of which can be emitted as object code.

Specialization

This implementation model lends itself to optimization when we know the specific argument types that will be used when invoking the generic function. In this case, some or all of the vtables provided for the constraints will effectively be constants. By specializing the generic function (at compile-time, link-time, or (if we have a JIT) run-time) for these types, we can eliminate the cost of the virtual dispatch, inline calls when appropriate, and eliminate the overhead of the generic system. Such optimizations can be performed based on heuristics, user direction, or profile-guided optimization.

An internal @_specialize function attribute allows developers to force full specialization by listing concrete type names corresponding to the function's generic signature. A function's generic signature is a concatenation of its generic context and the function's own generic type parameters.:

```
struct S<T> {
  var x: T
  @_specialize(where T == Int, U == Float)
  mutating func exchangeSecond<U>(_ u: U, _ t: T) -> (U, T) {
    x = t
    return (u, x)
  }
}

// Substitutes: <T, U> with <Int, Float> producing:
// S<Int>::exchangeSecond<Float>(u: Float, t: Int) -> (Float, Int)
```

@_specialize currently acts as a hint to the optimizer, which generates type checks and code to dispatch to the specialized routine without affecting the signature of the generic function. The intention is to support efforts at evaluating the performance of specialized code. The performance impact is not guaranteed and is likely to change with the optimizer. This attribute should only be used in conjunction with rigorous performance analysis. Eventually, a similar attribute could be defined in the language, allowing it to be exposed as part of a function's API. That would allow direct dispatch to specialized code without type checks, even across modules.

The exact syntax of the @_specialize function attribute is defined as:

```
@_specialize(exported: true, kind: full, where K == Int, V == Int)
@_specialize(exported: false, kind: partial, where K: _Trivial64)
func dictFunction<K, V>(dict: Dictionary<K, V>) {
}
```

If 'exported' is set, the corresponding specialization would have a public visibility and can be used by clients. If 'exported' is omitted, it's value is assumed to be 'false'.

If 'kind' is 'full' it means a full specialization and the compiler will produce an error if you forget to specify the type for some of the generic parameters in the 'where' clause. If 'kind' is 'partial' it means a partial specialization. If 'kind' is omitted, its value is assumed to be 'full.

The requirements in the where clause may be same-type constraints like 'T == Int', but they may also specify so-called layout constraints like 'T: _Trivial'.

The following layout constraints are currently supported:

- AnyObject - the actual parameter should be an instance of a class

- _NativeClass - the actual parameter should be an instance of a Swift native class
- _RefCountedObject - the actual parameter should be a reference-counted object
- _NativeRefCountedObject - the actual parameter should be a Swift-native reference-counted object
- _Trivial - the actual parameter should be of a trivial type, i.e. a type without any reference counted properties.
- _Trivial(SizeInBits) - like _Trivial, but the size of the type should be exactly 'SizeInBits' bits.
- _TrivialAtMost(SizeInBits) - like _Trivial, but the size of the type should be at most 'SizeInBits' bits.

Existential Types and Generics

Both existential types and generics depend on dynamic dispatching based on protocols. A value of an existential type (say, Comparable) is a pair (value, vtable). 'value' stores the current value either directly (if it fits in the 3 words allocated to the value) or as a pointer to the boxed representation (if the actual representation is larger than 3 words). By itself, this value cannot be interpreted, because its type is not known statically, and may change due to assignment. The vtable provides the means to manipulate the value, because it provides a mapping between the protocols to which the existential type conforms (which is known statically) to the functions that implements that functionality for the type of the value. The value, therefore, can only be safely manipulated through the functions in this vtable.

A value of some generic type T uses a similar implementation model. However, the (value, vtable) pair is split apart: values of type T contain only the value part (the 3 words of data), while the vtable is maintained as a separate value that can be shared among all T's within that generic function.

Overloading

Generic functions can be overloaded based entirely on constraints. For example, consider a binary search algorithm:

```
func binarySearch<
   C : EnumerableCollection where C.Element : Comparable
>(_ collection : C, value : C.Element)
  -> C.EnumeratorType
{
  // We can perform log(N) comparisons, but EnumerableCollection
  // only supports linear walks, so this is linear time
}

protocol RandomAccessEnumerator : Enumerator {
  // splits a range in half, returning both halves
  func split() -> (Enumerator, Enumerator)
}

func binarySearch<
   C : EnumerableCollection
    where C.Element : Comparable,
              C.EnumeratorType: RandomAccessEnumerator
>(_ collection : C, value : C.Element)
  -> C.EnumeratorType
{
  // We can perform log(N) comparisons and log(N) range splits,
  // so this is logarithmic time
}
```

If binarySearch is called with a sequence whose range type conforms to RandomAccessEnumerator, both of the generic functions match. However, the second function is more specialized, because its constraints are a superset of the constraints of the first function. In such a case, overloading should pick the more specialized function.

There is a question as to when this overloading occurs. For example, binarySearch might be called as a subroutine of another generic function with minimal requirements:

```
func doSomethingWithSearch<
  C : EnumerableCollection where C.Element : Ordered
>(
  _ collection : C, value : C.Element
) -> C.EnumeratorType
{
  binarySearch(collection, value)
}
```

At the time when the generic definition of doSomethingWithSearch is type-checked, only the first binarySearch() function applies, since we don't know that C.EnumeratorType conforms to RandomAccessEnumerator. However, when doSomethingWithSearch is actually invoked, C.EnumeratorType might conform to the RandomAccessEnumerator, in which case we'd be better off picking the second binarySearch. This amounts to run-time overload resolution, which may be desirable, but also has downsides, such as the potential for run-time failures due to ambiguities and the cost of performing such an expensive operation at these call sites. Of course, that cost could be mitigated in hot generic functions via the specialization mentioned above.

Our current proposal for this is to decide statically which function is called (based on similar partial-ordering rules as used in C++), and avoid run-time overload resolution. If this proves onerous, we can revisit the decision later.

Parsing Issues

The use of angle brackets to supply arguments to a generic type, while familiar to C++/C#/Java programmers, cause some parsing problems. The problem stems from the fact that '<', '>', and '>>' (the latter of which will show up in generic types such as Array<Array<Int>>) match the 'operator' terminal in the grammar, and we wish to continue using this as operators.

When we're in the type grammar, this is a minor inconvenience for the parser, because code like this:

```
var x : Array<Int>
```

will essentially parse the type as:

```
identifier operator identifier operator
```

and verify that the operators are '<' and '>', respectively. Cases involving <> are more interesting, because the type of:

```
var y : Array<Array<Int>>
```

is effectively parsed as:

```
identifier operator identifier operator identifier operator operator
```

by splitting the '>>' operator token into two '>' operator tokens.

However, this is manageable, and is already implemented for the (now deprecated) protocol composition syntax (protocol<>). The larger problem occurs at expression context, where the parser cannot disambiguate the tokens:

```
Matrix<Double>(10, 10)
```

i.e.,:

```
identifier operator identifier operator unspaced_lparen integer-literal comma integer-literal rparen
```

which can be interpreted as either:

```
(greater_than
  (less_than
    (declref Matrix)
    (declref Double)
  (tuple
    (integer_literal 10)
    (integer_literal 10)))
```

or:

```
(constructor Matrix<Double>
  (tuple
    (integer_literal 10)
    (integer_literal 10)))
```

Both Java and C# have this ambiguity. C# resolves the ambiguity by looking at the token after the closing '>' to decide which way to go; Java seems to do the same. We have a few options:

1. Follow C# and Java and implement the infinite lookahead needed to make this work. Note that we have true ambiguities, because one could make either of the above parse trees well-formed.
2. Introduce some kind of special rule for '<' like we have for '(', such as: an identifier followed by an unspaced '<' is a type, while an identifier followed by spacing and then '<' is an expression, or
3. Pick some syntax other than angle brackets, which is not ambiguous. Note that neither '(' nor '[' work, because they too have expression forms.
4. Disambiguate between the two parses semantically.

We're going to try a variant of #1, using a variation of the disambiguation rule used in C#. Essentially, when we see:

```
identifier <
```

we look ahead, trying to parse a type parameter list, until parsing the type parameter list fails or we find a closing '>'. We then look ahead an additional token to see if the closing '>' is followed by a '(', '.', or closing bracketing token (since types are most commonly followed by a constructor call or static member access). If parsing the type parameter list succeeds, and the closing angle bracket is followed by a '(', '.', or closing bracket token, then the '<...>' sequence is parsed as a generic parameter list; otherwise, the '<' is parsed as an operator.