Using Protocol Members with References to `Self` or `Self`-rooted Associated Types

Protocol requirements and protocol extension members may be accessed via a conformance constraint on a generic parameter, an opaque result type, or via the protocol type itself:

```
// An appropriately constrained generic parameter.
func foo<T: CustomStringConvertible>(arg: T) {
  let description: String = arg.description
}

do {
  // An appropriately constrained opaque result type.
  func foo() -> some CustomStringConvertible { true }

  let description: String = foo().description
}

// The protocol type.
func foo(arg: CustomStringConvertible) {
  let description: String = arg.description
}
```

While the former two options enable full access to the protocol interface, not all members may be accessible when the protocol is used as a type and not a constraint. Specifically, a protocol member cannot be accessed on a protocol type when its type signature contains a reference to `Self` or a `Self`-rooted associated type. Accessing such members on a protocol type is not supported because today the compiler does not have a well-defined meaning and means of representation for `Self` and `Self`-rooted associated types with respect to a protocol type `P`. As a result, the following code is not allowed:

```
protocol Shape {
  func matches(_ other: Self) -> Bool
}

func foo(_ shape: Shape) {
  // error: member 'matches' cannot be used on value of protocol type 'Shape'; use a generic constraint instead
  shape.matches(shape)
}

func foo(_ arg: Identifiable) {
  // error: member 'id' cannot be used on value of protocol type 'Identifiable'; use a generic constraint instead
  _ = arg.id
}
```

An exception to this limitation are members that contain `Self` only in [covariant](https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)) position (such as a method result type), where `Self` can be safely substituted with the protocol or protocol composition type used to access the member — a representable supertype. On the other hand, resorting to this ploy in contravariant parameter type position, like allowing one to pass a type-erased value to a method that accepts `Self`, is not type-safe and would expose the opportunity to pass in an argument of non-matching type.

```
protocol Shape {
  func duplicate() -> Self
}

func duplicateShape(_ shape: Shape) -> Shape {
  return shape.duplicate // OK, produces a value of type 'Shape'
}
```

Most use cases involving usage of protocol members that fall under the above restriction can instead be supported by constrained generics, opaque result types, or manual type-erasing wrappers. To learn more, see the sections on protocols, generics, and opaque types in the Language Guide. For a better understanding of existential types in particular, and an in-depth exploration of the relationships among these built-in abstraction models, we recommend reading the design document for improving the UI of the generics model.