Underlying Type Inference for Opaque Result Types

Opaque result types are a useful tool for abstracting the return type of a function or subscript, or type of a property. Although the concrete underlying type of an opaque type is hidden from clients, it is still inferred by the compiler, which enforces certain usage requirements:

- Property declarations with opaque types must have an initializer expression or getter, and functions or subscripts returning opaque types must have at least one `return` statement:

```
let x: some Equatable // error: property declares an opaque return type, but has no initializer expression from which to infer an underlying type
let y: some Equatable = 42 // OK
let z: some Equatable { // Also OK
  return "hello, " + "world!"
}

func foo() -> some Equatable { // error: function declares an opaque return type, but has no return statements in its body from which to infer an underlying type
  fatalError("Unimplemented")
}

func bar() -> some Equatable { // OK
  fatalError("Unimplemented")
  return 42
}
```

- The underlying type of an opaque type must be unique. In other words, if a function or subscript returns an opaque type, it must return values of the same underlying type from every `return` statement in its body.

```
func foo(bar: Bool) -> some Equatable { // error: function declares an opaque return type, but the return statements in its body do not have matching underlying types
  if bar {
    return "hello, world!" // note: return statement has underlying type 'String'
  } else {
    return 1 // note: return statement has underlying type 'Int'
  }
}

func bar(baz: Bool) -> some Equatable { // OK, both branches of the if statement return a value of the same underlying type, Int.
  if baz {
    return 100
  } else {
    return 200
  }
}
```

- Functions returning opaque types may be recursive. However, such functions must have at least one `return` statement that returns a concrete underlying type as opposed to the function's own opaque result type. Additionally, recursive calls may not be used to create an infinitely recursive opaque type.

```
func foo(_ x: Int) -> some Equatable { // error: function declares an opaque return
type, but has no return statements in its body from which to infer an underlying
type
  // Not allowed because there aren't any non-recursive returns to infer the
underlying type from.
  return foo(x+1)
}

struct EquatableWrapper<T: Equatable>: Equatable { var value: T }
func foo() -> some Equatable { // error: function opaque return type was inferred as
'EquatableWrapper<some Equatable>', which defines the opaque type in terms of itself
  // Not allowed because the use of EquatableWrapper creates an infinitely recursive
underlying type: EquatableWrapper<EquatableWrapper<EquatableWrapper<...>>>...>
  return EquatableWrapper(value: foo())
}

func bar(_ x: Int) -> some Equatable { // OK, the underlying type can be inferred
from the second return statement.
  if x > 0 {
    return bar(x-1)
  } else {
    return x
  }
}
```

To learn more about opaque result types, see the Opaque Types section of *The Swift Programming Language*.