orphan:

In-Place Operations

Author: Dave Abrahams
Author: Joe Groff
Abstract: The goal of efficiently processing complex data structures leads naturally to pairs of related operations such as + and +=: one that produces a new value, and another that mutates on the data structure in-place. By formalizing the relationship and adding syntactic affordances, we can make these pairs easier to work with and accelerate the evaluation of some common expressions.

Examples

In recent standard library design meetings about the proper API for sets, it was decided that the canonical `Set` interface should be written in terms of methods: [1]

```
struct Set<Element> {
  public func contains(_ x: Element) -> Bool                // x ∈ A, A ∋ x
  public func isSubsetOf(_ b: Set<Element>) -> Bool         // A ⊆ B
  public func isStrictSubsetOf(_ b: Set<Element>) -> Bool   // A ⊂ B
  public func isSupersetOf(_ b: Set<Element>) -> Bool       // A ⊇ B
  public func isStrictSupersetOf(_ b: Set<Element>) -> Bool // A ⊃ B
  ...
}
```

When we started to look at the specifics, however, we ran into a familiar pattern:

```
...
  public func union(_ b: Set<Element>) -> Set<Element>        // A ∪ B
  public mutating func unionInPlace(_ b: Set<Element>)        // A ∪= B

  public func intersect(_ b: Set<Element>) -> Set<Element>    // A ∩ B
  public mutating func intersectInPlace(_ b: Set<Element>)    // A ∩= B

  public func subtract(_ b: Set<Element>) -> Set<Element>     // A - B
  public mutating func subtractInPlace(_ b: Set<Element>)     // A -= B

  public func exclusiveOr(_ b: Set<Element>) -> Set<Element>  // A ⊕ B
  public mutating func exclusiveOrInPlace(_ b: Set<Element>)  // A ⊕= B
```

We had seen the same pattern when considering the API for `String`, but in that case, there are no obvious operator spellings in all of Unicode. For example:

```
struct String {
  public func uppercase() -> String
  public mutating func uppercaseInPlace()

  public func lowercase() -> String
  public mutating func lowercaseInPlace()

  public func replace(
    _ pattern: String, with replacement: String) -> String
  public mutating func replaceInPlace(
    _ pattern: String, with replacement: String)

  public func trim() -> String
  public mutating func trimInPlace()
  ...
}
```

It also comes up with generic algorithms such as `sort()` (which is mutating) and `sorted()`, the corresponding non-mutating version.

We see at least four problems with this kind of API:

1. The lack of a uniform naming convention is problematic. People have already complained about the asymmetry between mutating `sort()`, and non-mutating `reverse()`. The pattern used by `sort()` and `sorted()` doesn't apply everywhere, and penalizes the non-mutating form, which should be the more economical of the two.
2. Naming conventions that work everywhere and penalize the mutating form are awkward. In the case of `String` it was considered bad enough that we didn't bother with the mutating versions of any operations other than concatenation (which we spelled using + and +=).
3. Producing a complete interface that defines both variants of each operation is needlessly tedious. A working (if non-optimal) mutating version of `op(x: T, y: U) -> T` can always be defined as

   ```
   func opInPlace(x: inout T, y: U) {
     x = op(x, y)
   }
   ```

Default implementations in protocols could do a lot to relieve tedium here, but cranking out the same `xxxInPlace` pattern for each `xxx` still amounts to a lot of boilerplate.

4. Without formalizing the relationship between the mutating and non-mutating functions, we lose optimization opportunities. For example, it should be possible for the compiler to rewrite

```
let x = a.intersect(b).intersect(c).intersect(d)
```

as

```
var t = a.intersect(b)
t.intersectInPlace(c)
t.intersectInPlace(d)
let x = t
```

for efficiency, without forcing the user to sacrifice expressivity. This optimization would generalize naturally to more common idioms such as:

```
let newString = s1 + s2 + s3 + s4
```

Given all the right conditions, it is true that a similar optimization can be made at runtime for COW data structures using a uniqueness check on the left-hand operand. However, that approach only applies to COW data structures, and penalizes other cases.

The Proposal

Our proposal has four basic components:

1. Solve the naming convention problem by giving the mutating and non-mutating functions the same name.
2. Establish clarity at the point of use by extending the language to support a concise yet distinctive syntax for invoking the mutating operation.
3. Eliminate tedium by allowing mutating functions to be automatically generated from non-mutating ones, and, for value types, vice-versa (doing this for reference types is problematic due to the lack of a standard syntax for copying the referent).
4. Support optimization by placing semantic requirements on mutating and non-mutating versions of the same operation, and allowing the compiler to make substitutions.

Use One Simple Name

There should be one simple name for both in-place and non-mutating sorting: `sort`. Set union should be spelled `union`. This unification bypasses the knotty problem of naming conventions and makes code cleaner and more readable.

When these paired operations are free functions, we can easily distinguish the mutating versions by the presence of the address-of operator on the left-hand side:

```
let z = union(x, y)  // non-mutating
union(&x, y)         // mutating
```

Methods are a more interesting case, since on mutating methods, `self` is *implicitly* `inout`:

```
x.union(y) // mutating or non-mutating?
```

We propose to allow method pairs of the form:

```
extension X {
  func f(p0: T0, p1: T1, p2: T2, ...pn: Tn) -> X

  func =f(p0: T0, p1: T1, p2: T2, ...pn: Tn) -> Void
}
```

The second `=f` method is known as an **assignment method** [2]. Assignment methods are implicitly `mutating`. Together these two methods, `f` and `=f`, are known as an **assignment method pair**. This concept generalizes in obvious ways to pairs of generic methods, details open for discussion.

An assignment method is only accessible via a special syntax, for example:

```
x.=union(y)
```

The target of an assignment method is always required, even when the target is `self`:

```
extension Set {
  mutating func frob(_ other: Set) {
    let brick = union(other) // self.union(other) implied
    self.=union(other)       // calls the assignment method
    union(other)             // warning: result ignored
  }
}
```

Assignment Operator Pairs

Many operators have assignment forms, for instance, + has +=, - has -=, and so on. However, not all operators do; != is not the assignment form of !, nor is <= the assignment form of <. Operators with assignment forms can declare this fact in their `operator` declaration:

```
infix operator + {
  has_assignment
}
```

For an operator *op* which `has_assignment`, a pair of operator definitions of the form:

```
func op(X, Y) -> X

func op=(inout X, Y) -> Void
```

is known as an **assignment operator pair**, and similar generalization to pairs of generic operators is possible.

To avoid confusion, the existing `assignment` operator modifier, which indicates that an operator receives one of its operands implicitly `inout`, shall be renamed `mutating`, since it can also be applied to non-assignment operators:

```
postfix operator ++ {
  mutating // formerly "assignment"
}
```

If an operator `op` which `has_assignment` is in scope, it is an error to declare `op=` as an independent operator:

```
operator 💩 { has_assignment }

// Error: '💩=' is the assignment form of existing operator '💩'
operator 💩= { has_assignment }
```

Eliminating Tedious Boilerplate

Generating the In-Place Form

Given an ordinary method of a type `X`:

```
extension X {
  func f(p0: T0, p1: T1, p2: T2, ...pn: Tn) -> X
}
```

if there is no corresponding *assignment method* in `X` with the signature

```
extension X {
  func =f(p0: T0, p1: T1, p2: T2, ...pn: Tn) -> Void
}
```

we can compile the statement

```
x.=f(a0, p1: a1, p2: a2, ...pn: an)
```

as though it were written:

```
x = x.f(a0, p1: a1, p2: a2, ...pn: an)
```

Generating the Non-Mutating Form

Given an *assignment method* of a value type `X`:

```
extension X {
  func =f(p0: T0, p1: T1, p2: T2, ...pn: Tn) -> Void
}
```

if there is no method in `X` with the signature

```
extension X {
  func f(p0: T0, p1: T1, p2: T2, ...pn: Tn) -> X
}
```

we can compile the expression

```
x.f(a0, p1: a1, p2: a2, ...pn: an)
```

as though it were written:

```
{
  (var y: X) -> X in
  y.=f(a0, p1: a1, p2: a2, ...pn: an)
  return y
}(x)
```

Generating Operator Forms

If only one member of an *assignment operator pair* is defined, similar rules allow the generation of code using the other member. E.g.

we can compile

```
x op= expression
```

as though it were written:

```
x = x op (expression)
```

or

```
x op expression
```

as though it were written:

```
{
  (var y: X) -> X in
  y op=expression
  return y
}(x)
```

Class Types

Assignment and operators are generally applied to value types, but it's reasonable to ask how to apply them to class types. The first and most obvious requirement, in our opinion, is that immutable class types, which are fundamentally values, should work properly.

An assignment operator for an immutable class `X` always has the form:

```
func op= (lhs: inout X, rhs: Y) {
  lhs = expression creating a new X object
}
```

or, with COW optimization:

```
func op= (lhs: inout X, rhs: Y) {
  if isUniquelyReferenced(&lhs) {
    lhs.mutateInPlace(rhs)
  }
  else {
    lhs = expression creating a new X object
  }
}
```

Notice that compiling either form depends on an assignment to `lhs`.

A method of a class, however, cannot assign to `self`, so no explicitly-written assignment method can work properly for an immutable class. Therefore, at *least* until there is a way to reseat `self` in a method, explicitly-written assignment methods must be banned for class types:

```
// Invalid code:
class Foo {
  let x: Int
  required init(x: Int) { self.x = x }

  func advanced(_ amount: Int) -> Self {
    return Self(x: self.x + amount)
  }

  // Error, because we can't reseat self in a class method
  func =advanced(amount: Int) {
    self = Self(x: self.x + amount)
    // This would also be inappropriate, since it would violate value
    // semantics:
    // self.x += amount
  }
}
```

That said, given an explicitly-written non-assignment method that produces a new instance, the rules given above for implicitly-generated assignment method semantics work just fine:

```
// Valid code:
class Foo {
  let x: Int
  required init(x: Int) { self.x = x }

  func advanced(_ amount: Int) -> Self {
    return Self(x: self.x + amount)
```

```
  }
}

var foo = Foo(x: 5)
// Still OK; exactly the same as foo = foo.advanced(10)
foo.=advanced(10)
```

The alternative would be to say that explicitly-written assignment methods cannot work properly for immutable classes and "work" with reference semantics on other classes. We consider this approach indefensible, especially when one considers that operators encourage writing algorithms that can only work properly with value semantics and will show up in protocols.

Assignment Methods and Operators In Protocols

The presence of a `=method` signature in the protocol implies that the corresponding non-assignment signature is available. Declaring `=method` in a protocol generates two witness table slots, one for each method of the implied pair. If the `=method` signature is provided in the protocol, any corresponding non-assignment `method` signature is ignored. A type can satisfy the protocol requirement by providing either or both members of the pair; a thunk for the missing member of the pair is generated as needed.

When only the non-assignment `method` member of a pair appears in the protocol, it generates only one witness table slot. The assignment signature is implicitly available on existentials and archetypes, with the usual implicitly-generated semantics.

---

[1] Unicode operators, which dispatch to those methods, would also be supported. For example,

```
public func ⊃ <T>(a: Set<T>, b: Set<T>) -> Bool {
  return a.isStrictSupersetOf(b)
}
```

however we decided that these operators were sufficiently esoteric, and also inaccessible using current programming tools, that they had to remain a secondary interface.

[2] the similarity to getter/setter pairs is by no means lost on the authors. However, omitting one form in this case has a very different meaning than in the case of getter/setter pairs.