orphan:

Unified Function Syntax via Selector Splitting

> **Warning**
>
> This document was used in planning Swift 1.0; it has not been kept up to date and does not describe the current or planned behavior of Swift. In particular, we experimented with preposition-based splitting and decided against it.

Contents

Cocoa Selectors

A Cocoa selector is intended to convey what a method does or produces as well as what its various arguments are. For example, `NSTableView` has the following method:

```
- (void)moveRowAtIndex:(NSInteger)oldIndex toIndex:(NSInteger)newIndex;
```

Note that there are three pieces of information in the selector `moveRowAtIndex:toIndex:`:

1. What the method is doing ("moving a row").
2. What the first argument is ("the index of the row we're moving").
3. What the second argument is ("the index we're moving to").

However, there are only two selector pieces: "moveRowAtIndex" and "toIndex". The first selector piece is conveying both #1 and #2, and it reads well in English because the preposition "at" separates the action (`moveRow`) from the first argument (`AtIndex`), while the second selector piece conveys #3. Cocoa conventions in this area are fairly strong, where the first selector piece describes what the operation is doing or produces, and well as what the first argument is, and subsequent selector pieces describe the remaining arguments.

Splitting Selectors at Prepositions

When importing an Objective-C selector, split the first selector piece into a base method name and a first argument name. The actual split will occur just before the last preposition in the selector piece, using camelCase word boundaries to identify words. The resulting method name is:

```
moveRow(atIndex:toIndex:)
```

where `moveRow` is the base name, `atIndex` is the name of the first argument (note that the 'a' has been automatically lowercased), and `toIndex` is the name of the second argument.

In the (fairly rare) case where there are two prepositions in the initial selector, splitting at the last preposition improves the likelihood of a better split, because the last prepositional phrase is more likely to pertain to the first argument. For example, `appendBezierPathWithArcFromPoint:toPoint:radius:` becomes:

```
appendBezierPathWithArc(fromPoint:toPoint:radius:)
```

If there are no prepositions within the first selector piece, the entire first selector piece becomes the base name, and the first argument is unnamed. For example `UIView`'s `insertSubview:atIndex:` becomes:

```
insertSubview(_:atIndex:)
```

where '_' is a placeholder for an argument with no name.

Calling Syntax

By splitting selectors into a base name and argument names, Swift's keyword-argument calling syntax works naturally:

```
tableView.moveRow(atIndex: i, toIndex: j)
view.insertSubview(someView, atIndex: i)
```

The syntax generalizes naturally to global and local functions that have no object argument, i.e.,:

```
NSMakeRange(location: loc, length: len)
```

assuming that we had argument names for C functions or a Swift overlay that provided them. It also nicely handles cases where argument names aren't available, e.g.,:

```
NSMakeRange(loc, len)
```

as well as variadic methods:

```
NSString(stringwithFormat: "%@ : %@", key, value)
```

Declaration Syntax

The existing "selector-style" declaration syntax can be extended to better support declaring functions with separate base names and first argument names, i.e.:

```
func moveRow atIndex(Int) toIndex(Int)
```

However, this declaration looks very little like the call site, which uses a parenthesized argument list, commas, and colons. Let's eliminate the "selector-style" declaration syntax entirely. We can use the existing ("tuple-style") declaration syntax to mirror the call syntax directly:

```
func moveRow(_ atIndex: Int, toIndex: Int)
```

Now, sometimes the argument name that works well at the call site doesn't work well for the body of the function. For example, splitting the selector for `UIView`'s `contentHuggingPriorityForAxis:` results in:

```
func contentHuggingPriority(_ forAxis: UILayoutConstraintAxis) -> UILayoutPriority
```

The name `forAxis` works well at the call site, but not within the function body. So, we allow one to specify the name of the parameter for the body of the function:

```
func contentHuggingPriority(forAxis axis: UILayoutConstraintAxis) -> UILayoutPriority {
  // use 'axis' in the body
}
```

One can use '_' in either the argument or parameter name position to specify that there is no name. For example:

```
func f(_ a: Int)  // no argument name; parameter name is 'a'
```

```
func g(b _: Int)  // argument name is 'b'; no parameter name
```

The first function doesn't support keyword arguments; it is what an imported C or C++ function would use. The second function supports a keyword argument (`b`), but the parameter is not named (and therefore cannot be used) within the body. The second form is fairly uncommon, and will presumably only to be used for backward compatibility.

Method Names

The name of a method in this scheme is determined by the base name and the names of each of the arguments, and is written as:

```
basename(param1:param2:param3:)
```

to mirror the form of declarations and calls, with types, arguments, and commas omitted. In code, one can refer to the name of a function just by its basename, if the context provides enough information to uniquely determine the method. For example, when uncurrying a method reference to a variable of specified type:

```
let f: (UILayoutConstraintAxis) -> UILayoutPriority = view.contentHuggingPriority
```

To refer to the complete method name, place the method name in backticks, as in this reference to an optional method in a delegate:

```
if let method = delegate.`tableView(_:viewForTableColumn:row:)` {
  // ...
}
```

Initializers

Objective-C `init` methods correspond to initializers in Swift. Swift splits the selector name after the `init`. For example, `NSView`'s `initWithFrame:` method becomes the initializer:

```
init(withFrame: NSRect)
```

There is a degenerate case here where the `init` method has additional words following `init`, but there is no argument with which to associate the information, such as with `initForIncrementalLoad`. This is currently handled by adding an empty tuple parameter to store the name, i.e.:

```
init(forIncrementalLoad:())
```

which requires the somewhat unfortunate initialization syntax:

```
NSBitmapImageRep(forIncrementalLoad:())
```

Fortunately, this is a relatively isolated problem: Cocoa and Cocoa Touch contain only four selectors of this form:

```
initForIncrementalLoad
initListDescriptor
initRecordDescriptor
initToMemory
```

With a number that small, it's easy enough to provide overlays.

Handling Poor Mappings

The split-at-last-preposition heuristic works well for a significant number of selectors, but it is not perfect. Therefore, we will introduce an attribute into Objective-C that allows one to specify the Swift method name for that Objective-C API. For example, by default, the `NSURL` method `+bookmarkDataWithContentsOfURL:error:` will come into Swift as:

```
class func bookmarkDataWithContents(ofURL bookmarkFileURL: NSURL, error: inout NSError) -> NSData
```

However, one can provide a different mapping with the `method_name` attribute:

```
+ (NSData *)bookmarkDataWithContentsOfURL:(NSURL *)bookmarkFileURL error:(NSError **)error __attribute__((method_name(bookmarkData(
```

This attribute specifies the Swift method name corresponding to that selector. Presumably, the `method_name` attribute will be wrapped in a macro supplied by Foundation, i.e.,:

```
#define NS_METHOD_NAME(Name) __attribute__((method_name(Name)))
```

For 1.0, it is not feasible to mark up the Objective-C headers in the various SDKs. Therefore, the compiler will contain a list of mapping from Objective-C selectors to Swift method names. Post-1.0, we can migrate these mappings to the headers.

A mapping in the other direction is also important, allowing one to associate a specific Objective-C selector with a method. For example, a Boolean property:

```
var enabled: Bool {
  @objc(isEnabled) get {
    // ...
  }

  set {
    // ...
  }
}
```

Optionality and Ordering of Keyword Arguments

A number of programming languages have keyword arguments in one form or another, including Ada, C#, Fortran 95, Lua, OCaml, Perl 6, Python, and Ruby. Objective-C and Smalltalk's use of selectors is roughly equivalent, in the sense that the arguments get names. The languages with keyword arguments (but not Objective-C and Smalltalk) all allow re-ordering of arguments at the call site, and many allow one to provide arguments positionally without their associated name at the call site. However, Cocoa APIs were designed based on the understanding that they would not be re-ordered, and the sentence structure of some selectors depends on that. To that end, a new attribute `call_arguments(strict)` can be placed on any function and indicates that keyword arguments are required and cannot be reordered in calls to that function, i.e.:

```
@call_arguments(strict)
func moveRow(_ atIndex:Int, toIndex:Int)
```

Swift's Objective-C importer will automatically add this to all imported Objective-C methods, so that Cocoa APIs will retain their sentence structure.

Removing `with` and `for` from Argument Names

The prepositions `with` and `for` are commonly used in the first selector piece to separate the action or result of a method from the first argument, but don't themselves convey much information at either the call or declaration site. For example, `NSColor`'s `colorWithRed:green:blue:alpha:` is called as:

```
NSColor.color(withRed: 0.5, green: 0.5, blue: 0.5, alpha: 1.0)
```

The `with` in this case feels spurious and makes `withRed` feel out of sync with `green`, `blue`, and `alpha`. Therefore, we will remove the `with` (or `for`) from any argument name, so that this call becomes:

```
NSColor.color(red: 0.5, green: 0.5, blue: 0.5, alpha: 1.0)
```

In addition to improving the call site, this eliminates the need to rename parameters as often at the declaration site, i.e., this:

```
class func color(withRed red: CGFloat, green: CGFloat, blue: CGFloat, alpha: CGFloat) -> NSColor
```

becomes:

```
class func color(_ red: CGFloat, green: CGFloat, blue: CGFloat, alpha: CGFloat) -> NSColor
```

Note that we only perform this removal for `with` and `for`; other prepositions tend to have important meaning associated with them, and are therefore not removed. For example, consider calls to the `NSImage` method `-drawInRect:fromRect:operation:fraction:` with the leading prepositions retained and removed, respectively:

```
image.draw(inRect: x, fromRect: x, operation: op, fraction: 0.5)
image.draw(rect: x, rect: y, operation: op, fraction: 0.5)
```

Here, dropping the leading prepositions is actively harmful, because we've lost the directionality provided by `in` and `from` in the first two arguments. `with` and `for` do not have this problem.

The second concern with dropping `with` and `for` is that we need to either specify or infer the prepositions when declaring a method. For example, consider the following initializer:

```
init(frame: CGRect)
```

How would the compiler know to insert the preposition "with" into the name when computing the selector, so that this maps to `initWithFrame:`? In many cases, where we're overriding a method or initializer from a superclass or we are implementing a method to conform to a protocol, the selector can be deduced from method/initializer in the superclass or protocol. In those cases where new API is being defined in Swift where the selector requires a preposition, one would use the `objc` attribute with a selector:

```
@objc(initWithFrame:)
init(frame: CGRect)
```

Imported Objective-C methods would have the appropriate `objc` attribute attached to them automatically.

Which Prepositions?

English has a large number of prepositions, and many of those words also have other rules as adjectives, adverbs, and so on. The following list, taken from The English Club, with poetic, archaic, and non-US forms removed, provided the starting point for the list of prepositions used in splitting. The **bolded** prepositions are used to split; notes indicate whether Cocoa uses this preposition as a preposition in any of its selectors, as well as any special circumstances that affect inclusion or exclusion from the list.

Preposition	In Cocoa?	Dropped?	Notes
Aboard	No		
About	No*		Used as an adjective
Above	Yes	No	
Across	No		
After	Yes	No	
Against	Yes*		Misleading when split
Along	Yes	No	
Alongside	Yes	No	
Amid	No		
Among	No		
Anti	No*		Used as an adjective
Around	No		
As	Yes	No	
Astride	No		
At	Yes	No	
Bar	No*		Used as a noun
Barring	No		
Before	Yes	No	
Behind	No		
Below	Yes	No	
Beneath	No		
Beside	No		
Besides	No		
Between	Yes		Not amenable to parameters
Beyond	No		
But	No		
By	Yes	No	
Circa	No		
Concerning	No		
Considering	No		
Counting	No*		Used as an adjective
Cum	No		
Despite	No		
Down	No*		Used as a noun
During	Yes*		Misleading when split
Except	No		
Excepting	No		
Excluding	No		
Following	Yes	No	
For	Yes	**Yes**	
From	Yes	No	
Given	Yes*	No	Never splits a selector
In	Yes	No	
Including	Yes*	No	Never splits a selector
Inside	Yes	No	
Into	Yes	No	
Less	No*		Always "less than"
Like	Yes*		Misleading when split
Minus	No		
Near	No		
Notwithstanding	No		
Of	Yes	No	
Off	No*		Used as a noun
On	Yes	No	
Onto	No		
Opposite	No		
Out	No*		Used as an adverb
Outside	Yes*		Misleading when split
Over	No*		Used as an adverb
Past	No		
Pending	No*		Used as an adjective
Per	Yes*		Misleading to split
Plus	No		Used as an adjective
Pro	No		
Regarding	No		

Respecting	No		
Round	No		
Save	No*		Used as adjective, verb
Saving	No*		Used as adjective
Since	Yes	No	
Than	No*		Always "greater than"
Through	Yes*		Misleading when split
Throughout	No		
To	Yes	No	
Toward	No		
Towards	No		
Under	No		
Underneath	No		
Unlike	No		
Until	Yes	No	
Unto	No		
Up	No*		Used as adjective
Upon	Yes*		Misleading when split
Versus	No		
Via	Yes	No	
With	Yes	**Yes**	
Within	Yes	No	
Without	Yes*		Misleading when split
Worth	No		