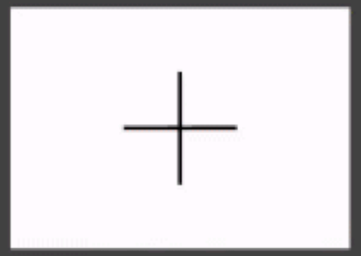

GlueKit
Composable Transformations of Observable Collections
Károly Lőrentey
@lorentey

Search for documents

Date | Name

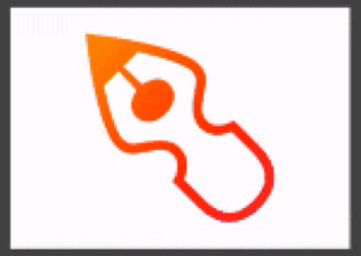


Create New

Lorem Ipsum
2016. Sep 29. 3:02:46

Dolor Sit Amet
2016. Sep 29. 3:02:43

Consectetur Adipiscing
2016. Sep 18. 17:04:05



Elit Sed Do
2016. Sep 2. 11:08:37

Eiusmod Tempor
2016. Sep 2. 11:05:03

Incididunt Ut Labore
2016. Sep 2. 3:43:47

Et Dolore Magna
2016. Sep 2. 1:13:36

Aliqua Ut Enim
2016. Sep 1. 14:17:53

Our task:

Maintain the contents of the UICollectionView, animating all changes

What's our Input? (1/2)

```
class FileMetadata {
    var displayName: String
    var modificationDate: Date
    var thumbnail: UIImage?
    ...
}

var documents: Set<FileMetadata>
```

The set of documents comes from an `NSMetadataQuery` or a `DispatchSource` watching the local documents folder.

What's our Input? (2/2)

```
var searchText: String?
var sortOrder: SortOrder

enum SortOrder {
    case byModificationDate
    case byName

    /// Returns true if `a` should be ordered before `b`.
    func comparator(a: FileMetadata, b: FileMetadata) -> Bool
}
```

What's the Output?

```
enum PickerItem {
    case newDocument
    case document(FileMetadata)

    func matches(text: String?) -> Bool {
        guard case .document(let file) = self else { return text == nil }
        guard let text = text else { return true }
        return file.displayName.localizedStandardContains(text)
    }
}

var items: [PickerItem]

class DocumentPickerCell: UICollectionViewCell {
    var item: PickerItem {
        didSet { ... }
    }
}
```

Let's Just Write a Pure Function

```
func generateItems(documents: Set<FileMetadata>,
                   searchText: String?,
                   sortOrder: SortOrder) -> [PickerItem] {
    return ([.newDocument] + documents.sorted(by: sortOrder.comparator)
                                      .map { .document($0) }
           ).filter { $0.matches(searchText) }
}
```

We have built the entire solution out of simple compositions of standard array and set transformations.

It's Time for Change

Idea: Handle Changes by Regenerating Everything

```
func updateItems() {
    let newItems = generateItems(documents, searchText, sortOrder)
    let delta = calculateDifference(from: self.items, to: newItems)
    self.items = newItems

    let cv = self.collectionView
    cv.performBatchUpdates {
        cv.deleteItems(at: delta.deletedIndices)
        cv.insertItems(at: delta.insertedIndices)
        for (from, to) in delta.movedIndices {
            cv.moveItem(at: from, to: to)
            if let cell = cv.cellForItem(at: from) as? DocumentPickerCell {
                cell.item = self.items[to]
            }
        }
    }
}
```

What a Waste!

Can't We Do Better?

```
func generateItems(documents: Set<FileMetadata>,
                   searchText: String?,
                   sortOrder: SortOrder) -> [PickerItem] {
    return ([.newDocument]
             + documents
                 .sorted(by: sortOrder.comparator)
                 .map { .document($0) }
            ).filter { $0.matches(searchText) }
}
```

It seems we might be able to transform the pure functional solution into a form that handles incremental changes, too

(Somehow, maybe)

```
func generateItems(documents: Set<FileMetadata>,
                   searchText: String?,
                   sortOrder: SortOrder) -> [PickerItem] {
    return (
        [.newDocument]
        + documents
            .sorted(by: sortOrder.comparator)
            .map { .document($0) }
    ).filter { $0.matches(searchText) }
}
```

Simple compositions of standard transformations on arrays and sets.

Spoiler Alert!

```
let documents: ObservableSet<FileMetadata>
let searchText: Observable<String?>
let sortOrder: Observable<SortOrder>

let items: ObservableArray<PickerItem> = (
    ObservableArray.constant([.newDocument])
    + documents
        .sorted(by: sortOrder.map { $0.comparator })
        .map { .document($0) }
).filter(where: searchText.map { text in { $0.matches(text) } })
```

Simple compositions of standard transformations on **observable** values, sets and arrays.

What's an observable value?

It's just a particular way to represent mutating state.

An **observable value** is an entity that

- has a getter for a **value** that may change from time to time
- provides an interface for **subscribing** to the value's **incremental change notifications**

Basically, it supports both pull- and push-based access to its value

Subscription & notification API

```
typealias Sink<Value> = (Value) -> Void
protocol SourceType {
    associatedtype Value
    func connect(_ sink: Sink<Value>) -> Connection
}
class Signal<Value>: SourceType {
    func connect(_ sink: Sink<Value>) -> Connection
    func send(_ value: Value)
}
let s = Signal<Int>()
let c = s.connect { print($0) }
s.send(42)      // Prints "42"
c.disconnect()
```

This is just the classic Observer pattern.

Incremental Change

```
protocol ChangeType {
    associatedtype Value

    init(from old: Value, to new: Value)

    func apply(on value: inout Value) // Partial fn!
    func merged(with change: Self) -> Self
    func reversed() -> Self
}
```

Abstract Observable

```
protocol ObservableType {
    associatedtype Change: ChangeType

    var value: Change.Value { get }
    var changes: Source<Change> { get }
}
```

This is a bit too abstract. We need to know more about the structure of the value and the details of the change description to do interesting operations on observables.

Our Family of Observables

We'll differentiate observables into three distinct flavors by the structure of their value types.

Value type:	T	Array<T>	Set<T>
Protocol name:	ObservableScalarType	ObservableArrayType	ObservableSetType
Change type:	ScalarChange<T>	ArrayChange<T>	SetChange<T>
Type-lifted:	Observable<T>	ObservableArray<T>	ObservableSet<T>
Concrete:	Variable<T>	ArrayVariable<T>	SetVariable<T>

Additional flavors (dictionaries, tree hierarchies etc.) are left as an exercise for the reader.

Observable Scalars

```
protocol ObservableScalarType: ObservableType {
    associatedtype Value

    var value: Value { get }
    var changes: Source<ScalarChange<Value>> { get }
}
struct ScalarChange<Value>: ChangeType {
    let old: Value
    let new: Value

    init(from old: Value, to new: Value) { self.old = old; self.new = new }
    func apply(on value: inout Value) { value = new }
    func merged(with change: ScalarChange) -> ScalarChange {
        return .init(from: old, to: change.new)
    }
    func reversed() -> ScalarChange { return .init(from: new, to: old) }
}
```

Concrete Scalar Observable: Variable

```
class Variable<Value>: ObservableScalarType {
    typealias Change = SimpleChange<Value>
    let signal = Signal<Change>()
    var value: Value {
        didSet {
            signal.send(Change(from: oldValue, to: newValue))
        }
    }
    var changes: Source<Change> { return signal.source }
    init(_ value: Value) { self.value = value }
}

let name = Variable<String>("Fred")
let connection = name.changes.connect { c in print("Bye \(c.old), hi \(c.new)!") }
name.value = "Barney"      // Prints "Bye Fred, hi Barney!"
connection.disconnect()
```

Observable Map

```
extension ObservableScalarType {
    func map<R>(_ transform: (Value) -> R) -> Observable<R> {
        return Observable(
            getter: { transform(self.value) },
            changes: { self.changes.map { ScalarChange(from: transform($0.old),
                                                       to: transform($0.new) } }
        )
    }
}

let quiet = Variable<String>("Fred")
let loud = quiet.map { "\($0.uppercased())!!!" }

print(loud.value)           // Prints "FRED!!!"
let c = loud.connect { print($0.new) }
quiet.value = "Barney"      // Prints "BARNEY!!!"
c.disconnect()
```

A Combination of Two Scalar Observables

```
class BinaryObservable<O1, O2, Value>: ObservableScalarType
where O1: ObservableScalarType, O2: ObservableScalarType {
    let o1: O1;       let o2: O2
    var v1: O1.Value; var v2: O2.Value
    let compose: (O1.Value, O2.Value) -> Value
    let signal = Signal<ScalarChange<Value>>()
    let c: [Connection] = []

    init(_ o1: O1, _ o2: O2, compose: (O1.Value, O2.Value) -> Value) {
        self.o1 = o1; self.o2 = o2; v1 = o1.value; v2 = o2.value; self.compose = compose
        self.c = [
            o1.connect { signal.send(ScalarChange(from: compose($0.old, self.v2),
                                                  to: compose($0.new, self.v2))) },
            o2.connect { signal.send(ScalarChange(from: compose(self.v1, $0.old),
                                                  to: compose(self.v1, $0.new))) }
        ]
    }
    var value: Value { return compose(v1, v2) }
    var changes: Source<ScalarChange<Value>> { return signal.source }
}
```

Everybody loves operator overloading

```
func + <O: ObservableScalarType>(a: O, b: O) -> Observable<O.Value>
where O.Value: IntegerArithmetic
    return BinaryObservable(a, b, +).observable
}

let a = Variable<Int>(23)
let b = Variable<Int>(42)

let sum = a + b       // Type is Observable<Int>
print(sum.value)      // Prints "65"
a.value = 13
print(sum.value)      // Prints "55"
```

Observable Expressions

```
typealias OST = ObservableScalarType

func + <O: OST>(a: O, b: O) -> Observable<O.Value> where O.Value: IntegerArithmetic
func - <O: OST>(a: O, b: O) -> Observable<O.Value> where O.Value: IntegerArithmetic
func * <O: OST>(a: O, b: O) -> Observable<O.Value> where O.Value: IntegerArithmetic
func / <O: OST>(a: O, b: O) -> Observable<O.Value> where O.Value: IntegerArithmetic

func == <O: OST>(a: O, b: O) -> Observable<Bool> where O.Value: Equatable
func != <O: OST>(a: O, b: O) -> Observable<Bool> where O.Value: Equatable
func < <O: OST>(a: O, b: O) -> Observable<Bool> where O.Value: Comparable
func <= <O: OST>(a: O, b: O) -> Observable<Bool> where O.Value: Comparable

prefix func ! <O: OST>(a: O) -> Observable<Bool> where O.Value == Bool
func && <O: OST>(a: O, b: O) -> Observable<Bool> where O.Value == Bool
func || <O: OST>(a: O, b: O) -> Observable<Bool> where O.Value == Bool

let predicate: Observable<Bool> = !(a > b && a + b < c)       // Neat (?)
```

Observable Arrays

```
protocol ObservableArrayType: ObservableType {
    associatedtype Element

    var count: Int { get }
    subscript(index: Int) -> Element { get }
    subscript(bounds: Range<Int>) -> Array<Element> { get }
    var value: Array<Element> { get }

    var changes: Source<ArrayChange<Element>> { get }
}
extension ObservableArrayType {
    var value: Array<Element> { return self[0 ..< count] }
    subscript(index: Int) -> Element {
        return self[index ..< index + 1].first!
    }
}
```

Array Changes (1/2)

```
enum ArrayModification<Element> {
    case insert(Element, at: Int)
    case remove(Element, at: Int)
    case replace(Element, at: Int, with: Element)
    case replaceRange([Element], at: Int, with: [Element])
}

extension Array {
    mutating func apply(_ mod: ArrayModification<Element>) { ... }
}
```

Array Changes (2/2)

```
struct ArrayChange<Element>: ChangeType {
    typealias Value = Array<Element>
    var modifications: [ArrayModification<Element>] // Sorted by index

    init()
    mutating func add(_ mod: ArrayModification<Element>)

    init(from old: [Element], to new: [Element])
    func apply(on value: inout [Element])
    func merged(with change: ArrayChange<Element>) -> ArrayChange<Element>
    func reversed() -> ArrayChange<Element>
}
```

ArrayChange Extensions

```
extension ArrayChange {
    func map<R>(_ transform: (Element) -> R) -> ArrayChange<R>
    func shift(by delta: Int) -> ArrayChange<Element>

    // For basic UITableView/UICollectionView animations
    var deletedIndices: IndexSet
    var insertedIndices: IndexSet
}

extension ArrayChange where Element: Hashable {
    // For complex UITableView/UICollectionView animations,
    // including detection of moved rows
    func batched() -> (deleted: IndexSet,
                       inserted: IndexSet,
                       moved: [(from: Int, to: Int)])
}
```

Array Transformations

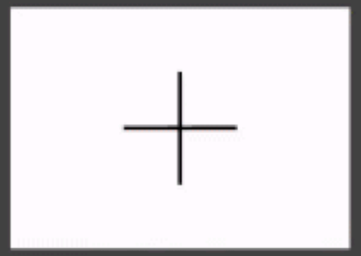
```
extension ObservableArrayType {
    func map<R>(_ transform: @escaping (Element) -> R) -> ObservableArray<R>

    func filtered(test: @escaping (Element) -> Bool) -> ObservableArray<Element>
    func filtered(test: Observable<(Element) -> Bool>) -> ObservableArray<Element>
    func filtered(test: @escaping (Element) -> Observable<Bool>) -> ObservableArray<Element>
}

// Concatenation
func + <O1, O2>(a: O1, b: O2) -> ObservableArray<O1.Element>
    where O1: ObservableArrayType, O2: ObservableArrayType, O1.Element == O2.Element
```

Search for documents

Date | Name

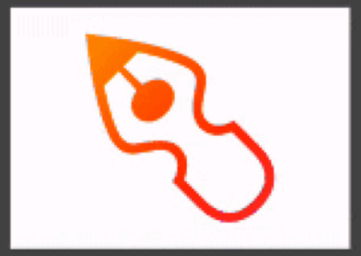


Create New

Lorem Ipsum
2016. Sep 29. 3:02:46

Dolor Sit Amet
2016. Sep 29. 3:02:43

Consectetur Adipiscing
2016. Sep 18. 17:04:05



Elit Sed Do
2016. Sep 2. 11:08:37

Eiusmod Tempor
2016. Sep 2. 11:05:03

Incididunt Ut Labore
2016. Sep 2. 3:43:47

Et Dolore Magna
2016. Sep 2. 1:13:36

Aliqua Ut Enim
2016. Sep 1. 14:17:53

```
class DocumentPickerController: UICollectionViewController {
    let documents: ObservableSet<FileMetadata> = ...
    let searchText: Observable<String?> = ...
    let sortOrder: Observable<SortOrder> = ...
    let items: ObservableArray<PickerItem>

    init(...) {
        self.items = (
            ObservableArray.constant([.newDocument])
            + documents
                .sorted(by: sortOrder.map { $0.comparator })
                .map { .document($0) }
        ).filter(where: searchText.map { text in { $0.matches(text) } })
    }
}
```

```
override func numberOfSections(in collectionView: UICollectionView) -> Int {
    return 1
}

override func collectionView(_ collectionView: UICollectionView,
                             numberOfItemsInSection section: Int) -> Int {
    return self.items.count
}

override func collectionView(_ collectionView: UICollectionView,
                             cellForItemAt indexPath: IndexPath)
    -> UICollectionViewCell {

    precondition(indexPath.section == 0)
    let cell = collectionView.dequeueReusableCell(
        withReuseIdentifier: PickerCell.reuseIdentifier,
        for: indexPath) as! PickerCell
    cell.item = self.items[indexPath.row]
    ...
    return cell
}
```

```
override func viewWillAppear(_ animated: Bool) {
    ...
    let cv = self.collectionView
    cv.reloadData()
    itemConnection = items.changes.connect { change in
        let batch = change.batched()
        cv.performBatchUpdates({
            cv.deleteItems(at: batch.deleted)
            cv.insertItems(at: batch.inserted)
            for (from, to) in batch.moved {
                cv.moveItem(at: from, to: to)
                if let cell = cv.cellForItem(at: from) as? Cell {
                    cell.item = self.items[to]
                }
            }
        }
    }
}
override func viewDidDisappear(_ animated: Bool) {
    ...
    itemConnection.disconnect()
}
```

Observable values lets us treat mutating state as if it wasn't mutating at all

You don't have to give up mutations to do functional-style programming

One more thing

```
extension ObservableSetType {
    func sorted<Field: ObservableType>(
        using key: (Element) -> Field,
        by comparator: (Field.Value, Field.Value) -> Bool)
    -> ObservableArray<Element>
}
```


Thank you!
https://github.com/lorentey/GlueKit
Károly Lőrentey
@lorentey

The Problem of Invalid Intermediate Values

```
let a = Variable<Int>(0)

let sum = a + a

let c = sum.connect { print("\($0.old) -> \($0.new)") }
a.value = 1         // Prints: "0 -> 1", "1 -> 2"
a.value = 3         // Prints: "2 -> 4", "4 -> 6"

c.disconnect()
```

One general solution is to convert change notifications into a two-phase system — willChange/didChange. (Does this sound familiar?)

Keypath Observing like Cocoa's KVO

```
extension ObservableArrayType {
    func selectEach<Field: ObservableScalarType>(_ key: @escaping (Element) -> Field)
        -> ObservableArray<Field.Value>
    func selectEach<Field: ObservableArrayType>(_ key: @escaping (Element) -> Field)
        -> ObservableArray<Field.Element>
}
```

Type-safe Keypath Observing

```
class Book { let title: Variable<String> }
class Bookshelf { let books: ArrayVariable<Book> }

let b1 = Book("Anathem")
let b2 = Book("Cryptonomicon")
let shelf: ArrayVariable<Book> = [b1, b2]

let titles = shelf.selectEach{$0.title}   // Type is ObservableArray<String>
let c = titles.changes.connect { _ in print(titles.value) }
print(titles.value)             // Prints "[Anathem, Cryptonomicon]"
b1.title = "Seveneves"          // Prints "[Seveneves, Cryptonomicon]"
shelf.append(Book("Zodiac"))    // Prints "[Seveneves, Cryptonomicon, Zodiac]"
shelf.remove(at: 1)             // Prints "[Seveneves, Zodiac]"
b2.title = "The Diamond Age"    // Nothing printed, b2 isn't in shelf
c.disconnect()
```

Observable Sets

```
protocol ObservableSetType: ObservableType {
    associatedtype Element: Hashable

    var count: Int { get }
    func contains(_ element: Element) -> Bool
    var value: Set<Element> { get }

    var changes: Source<SetChange<Element>> { get }
}

struct SetChange<Element: Hashable>: ChangeType {
    let removed: Set<Element>
    let inserted: Set<Element>
    ...
}
```

Operations on Observable Sets

```
extension ObservableSetType {
    func filtered(_ predicate: @escaping (Element) -> Bool)
        -> ObservableSet<Element>
    func filtered(_ predicate: @escaping (Element) -> Observable<Bool>)
        -> ObservableSet<Element>

    func sorted(by areInIncreasingOrder: @escaping (Element, Element) -> Bool)
        -> ObservableArray<Element>
    func sorted(by areInIncreasingOrder: Observable<(Element, Element) -> Bool>)
        -> ObservableArray<Element>
    func sorted<Field: ObservableType>(using key: (Element) -> Field,
                                       by comparator: (Field.Value, Field.Value) -> Bool)
        -> ObservableArray<Element>
}
```

It is also possible to define the observable union, intersection, difference, exclusiveOr, etc.