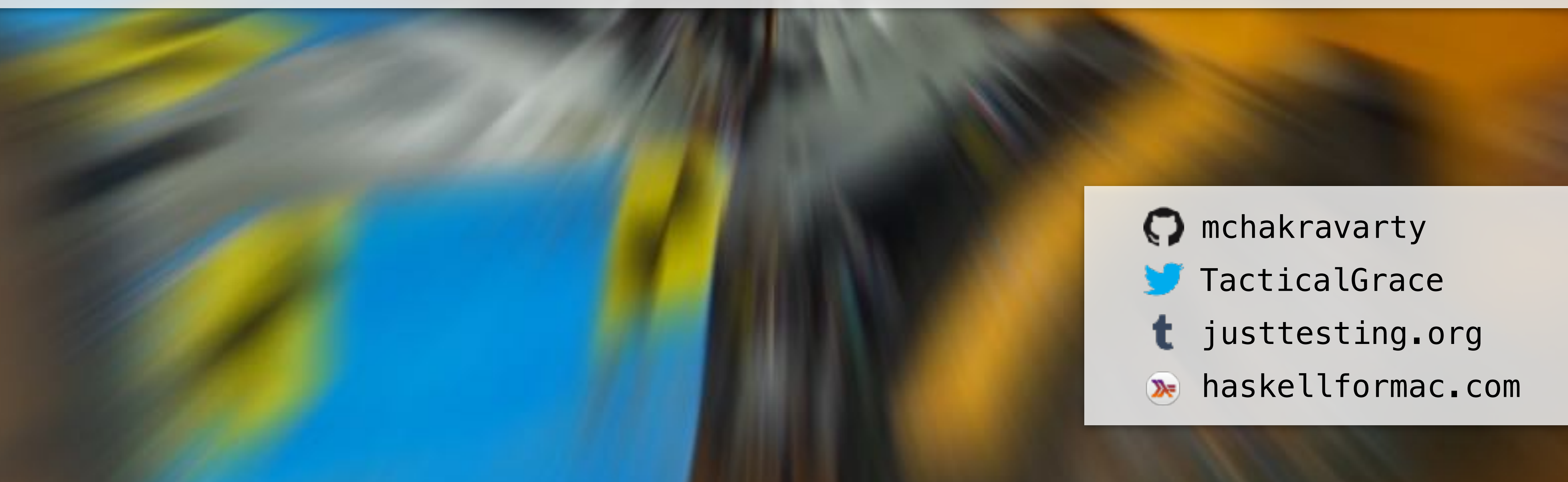
A Type is Worth a Thousand Tests

Manuel M T Chakravarty
Applicative &
Tweag I/O



Let's talk about…

Let's talk about…

Swift

Let's talk about…

Language Design

Swift

Let's talk about…

Language Design

Ecosystem

Swift

Let’s talk about…

Why is it extraordinary?

Language Design

Ecosystem

Unique Event

Java

C#

Objective-C

Java

Scala

Kotlin

C#

Objective-C

Java

Scala

Kotlin

C#

F#

Objective-C

Java

Scala

Kotlin

C#

F#

Objective-C

Swift

Why do languages become popular?

Why do languages become popular?

Platforms

Why do languages become popular?

Objective-C
Cocoa

JavaScript
Web

Python/Ruby
Cloud

Java
JVM

C#
.NET

C/C++
Unix

Platforms

Why do languages become popular?

Swift

Cocoa

JavaScript

Web

Python/Ruby

Cloud

Java

JVM

C#

.NET

C/C++

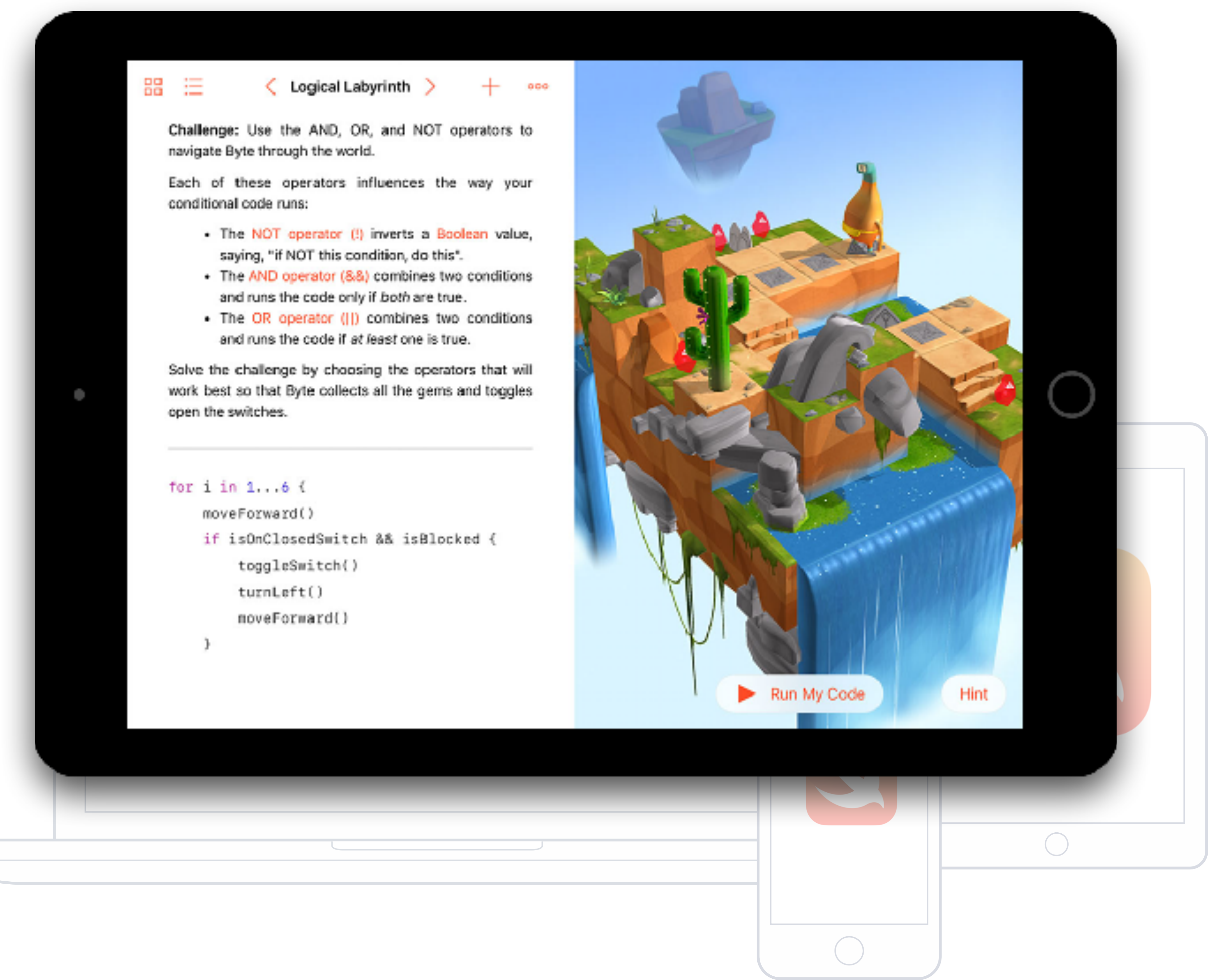
Unix

Platforms

Why will Swift be popular?

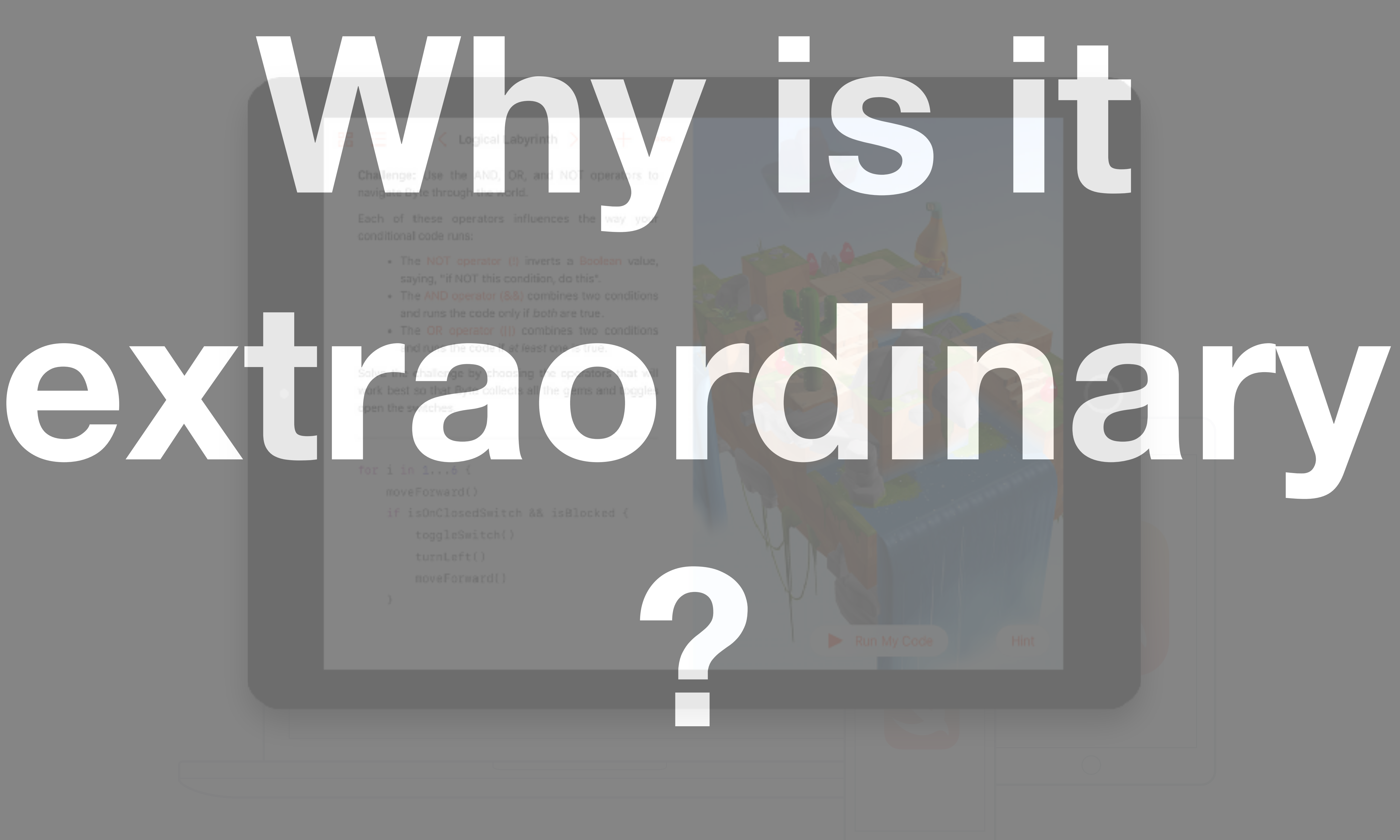
Why will Swift be popular?

Why will Swift be popular?



Why will Swift be popular?

Why is it extraordinary?



Functional Programming

From Objective-C to Swift

From Objective-C to Swift

Value types
(products & sums)

Advanced
type system

Explicit control of
mutability

Explicit
optionals

Generics &
protocols with
associated types

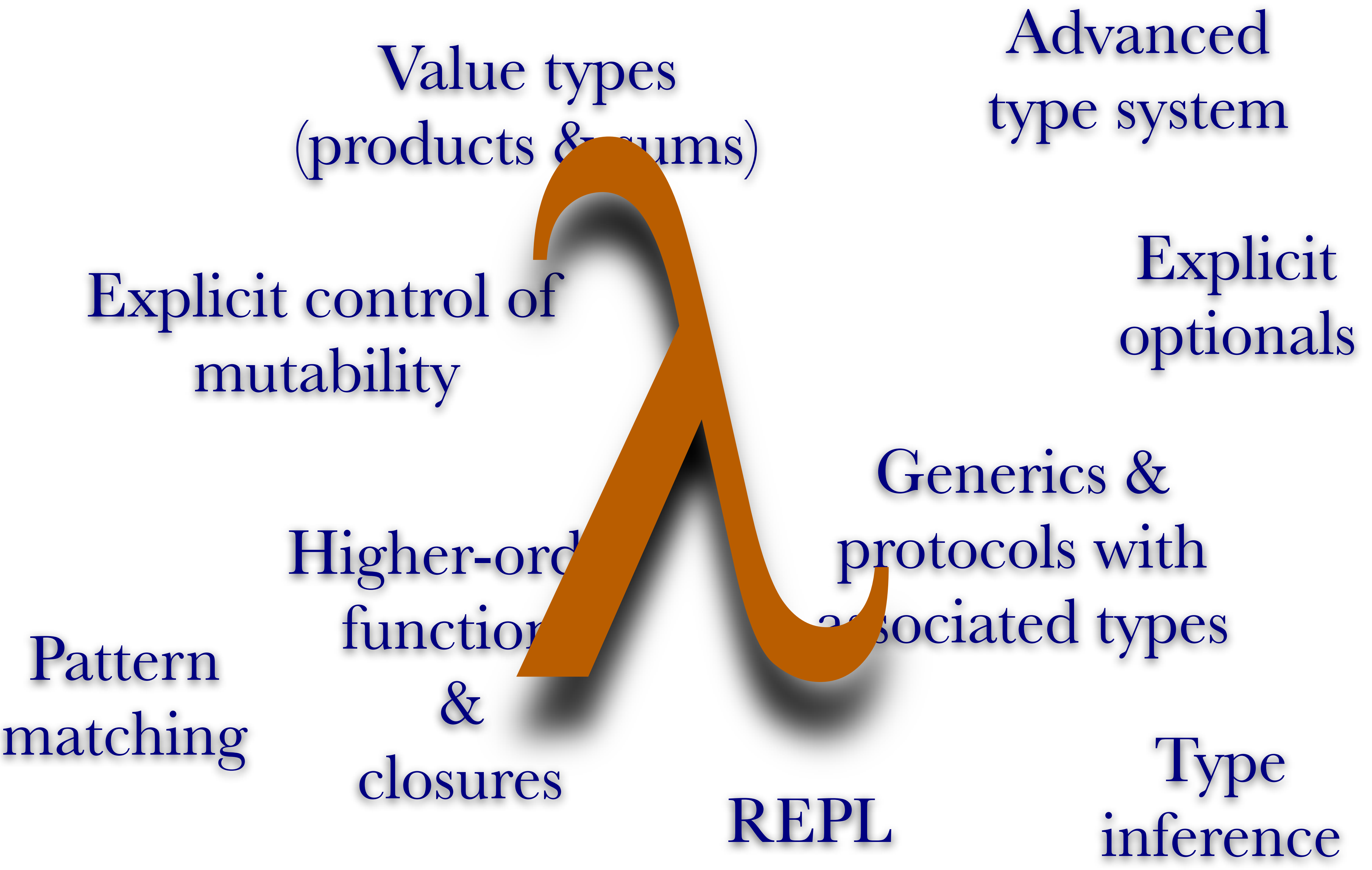
Pattern
matching

Higher-order
functions
&
closures

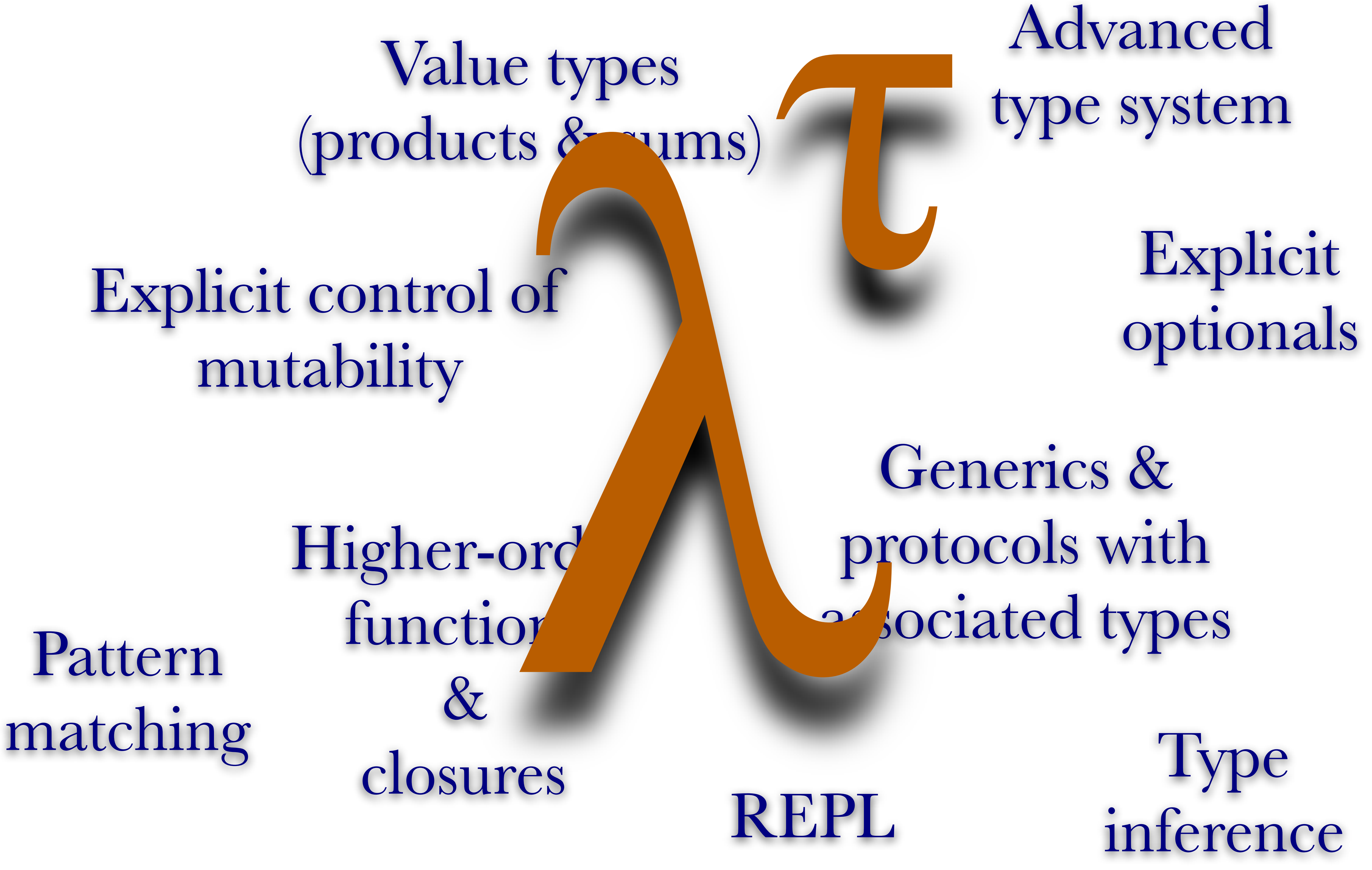
REPL

Type
inference

From Objective-C to Swift



From Objective-C to Swift



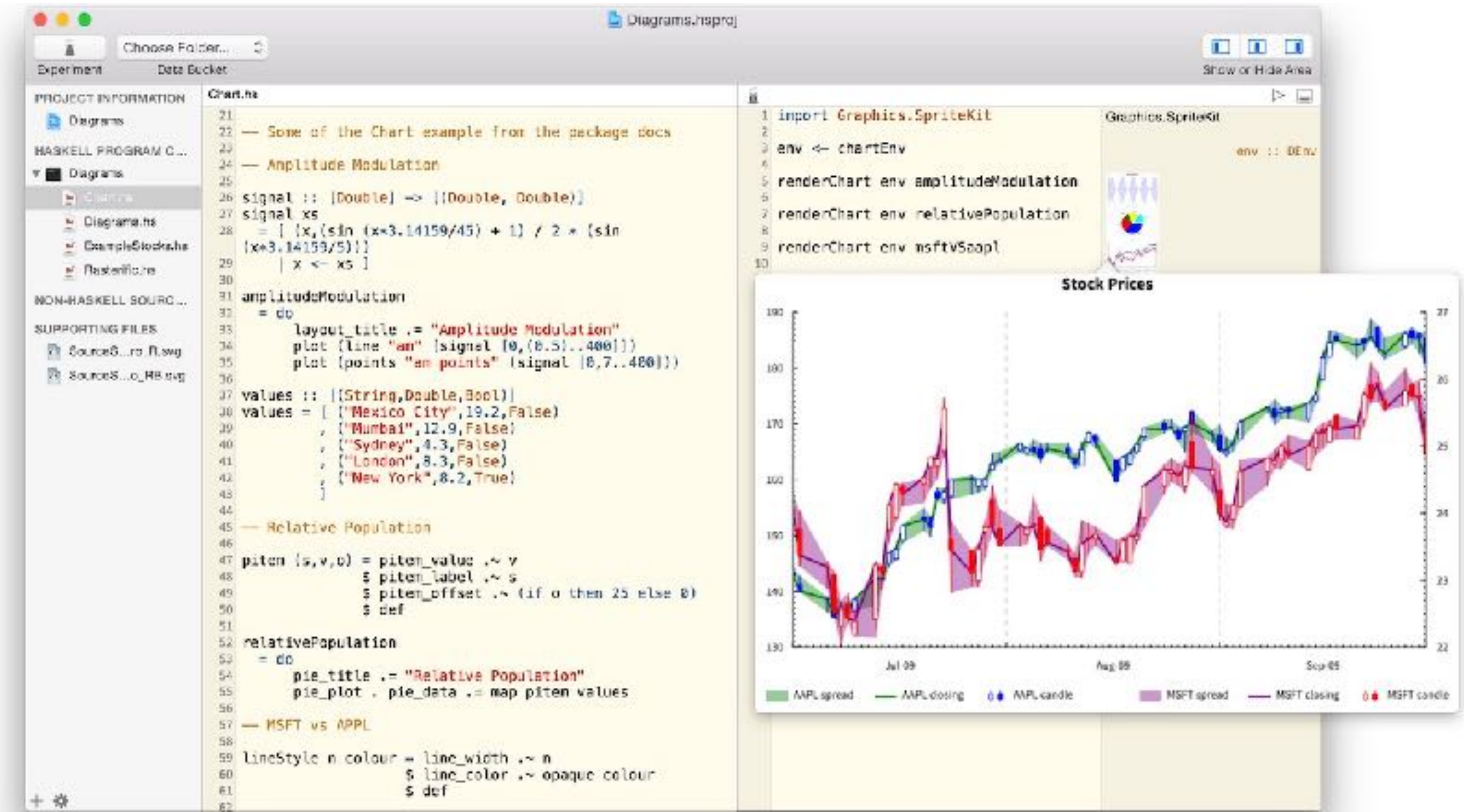
"Swift encourages typed functional programming in a mainstream language."

Haskell for Mac

Adopting Swift right out of the gate

Haskell for Mac

Adopting Swift right out of the gate

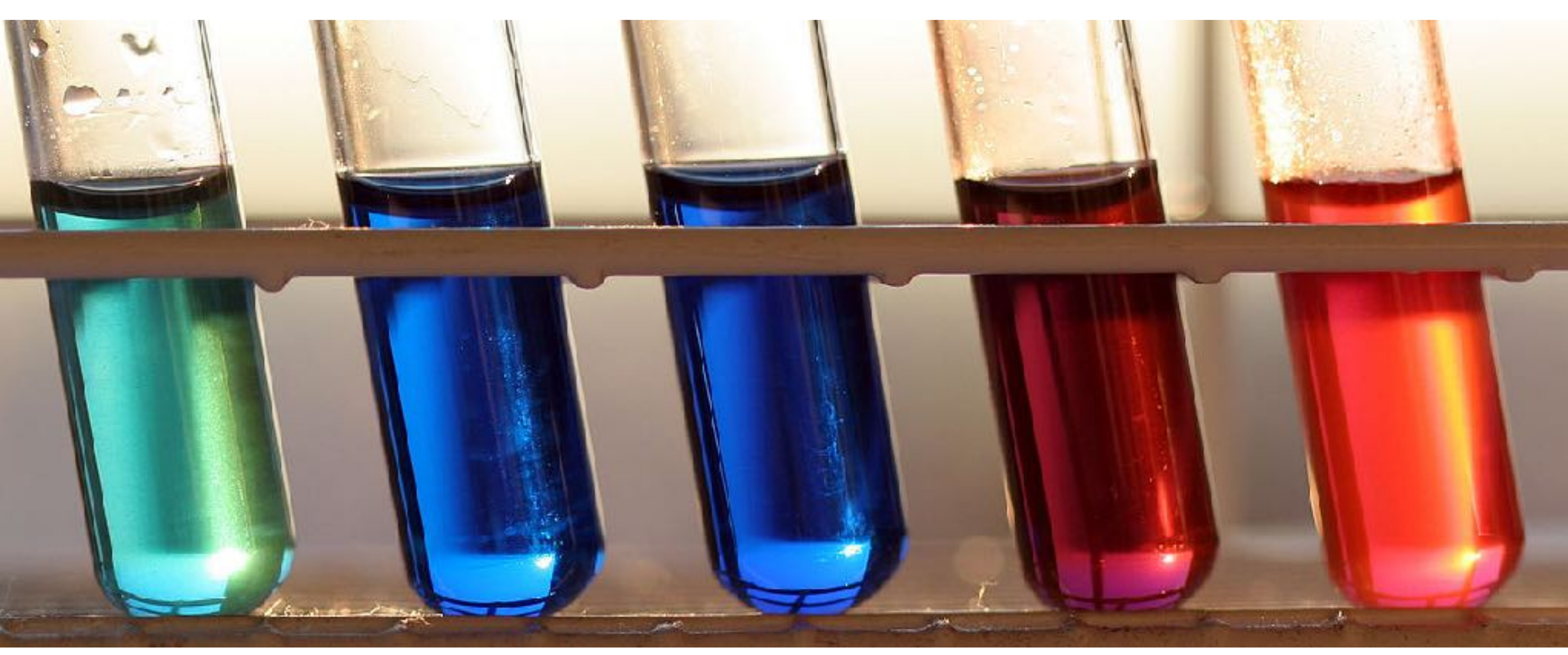


“Swift encourages typed functional programming in a mainstream language.”

Why Type Systems?

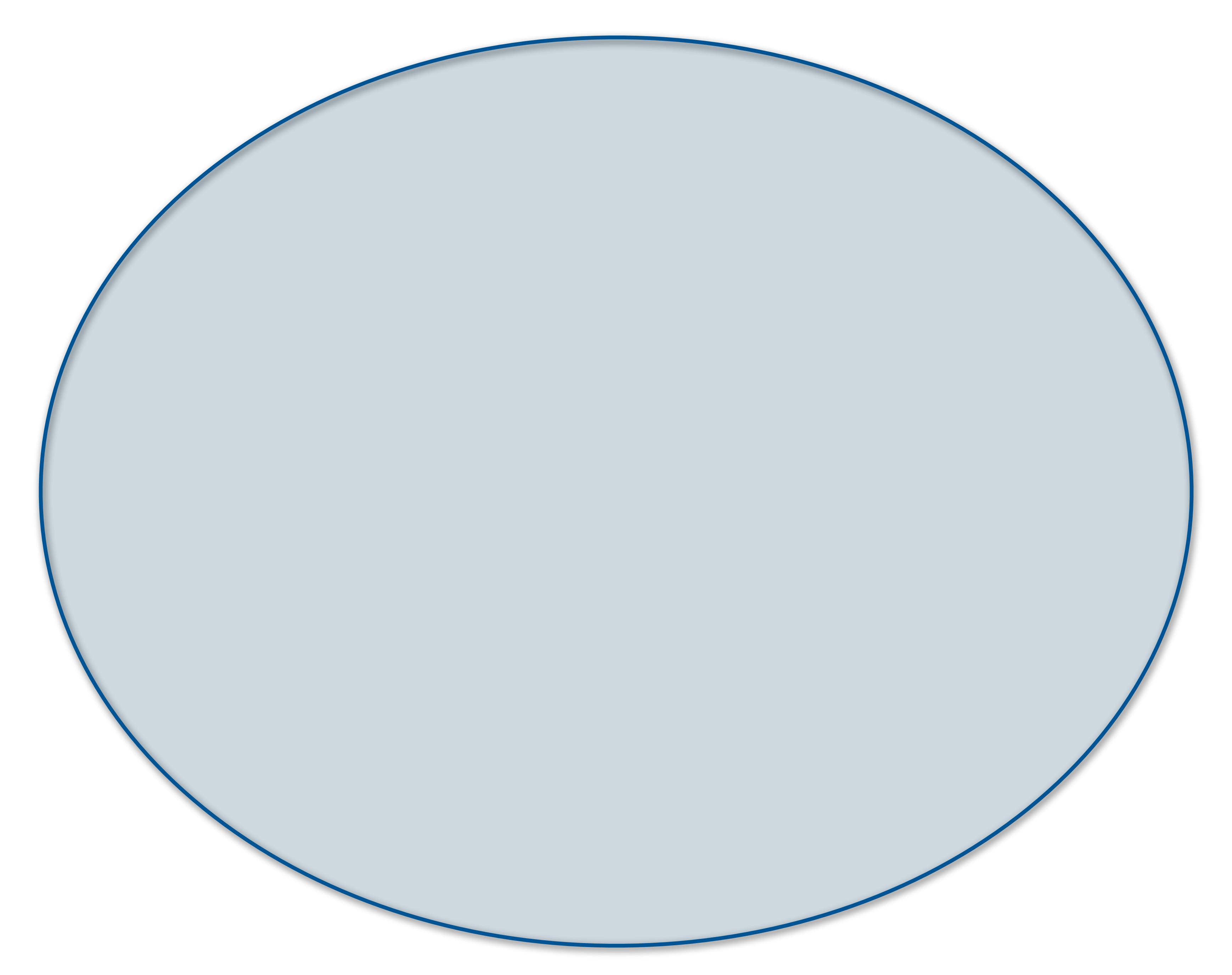
Catch bugs & prevent crashes

Less testing!

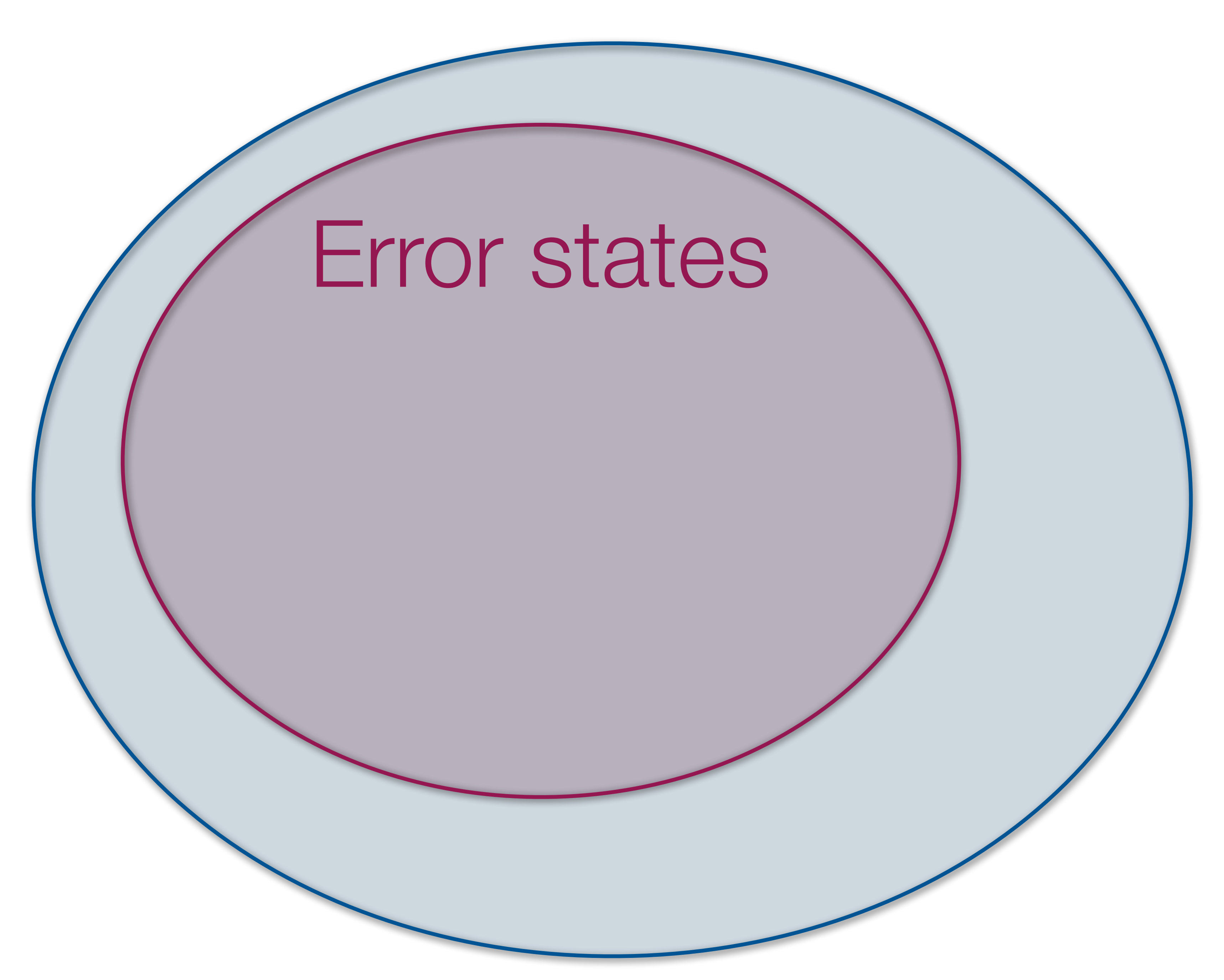


“What if you could gain the same confidence in correctness with fewer or simpler tests?”

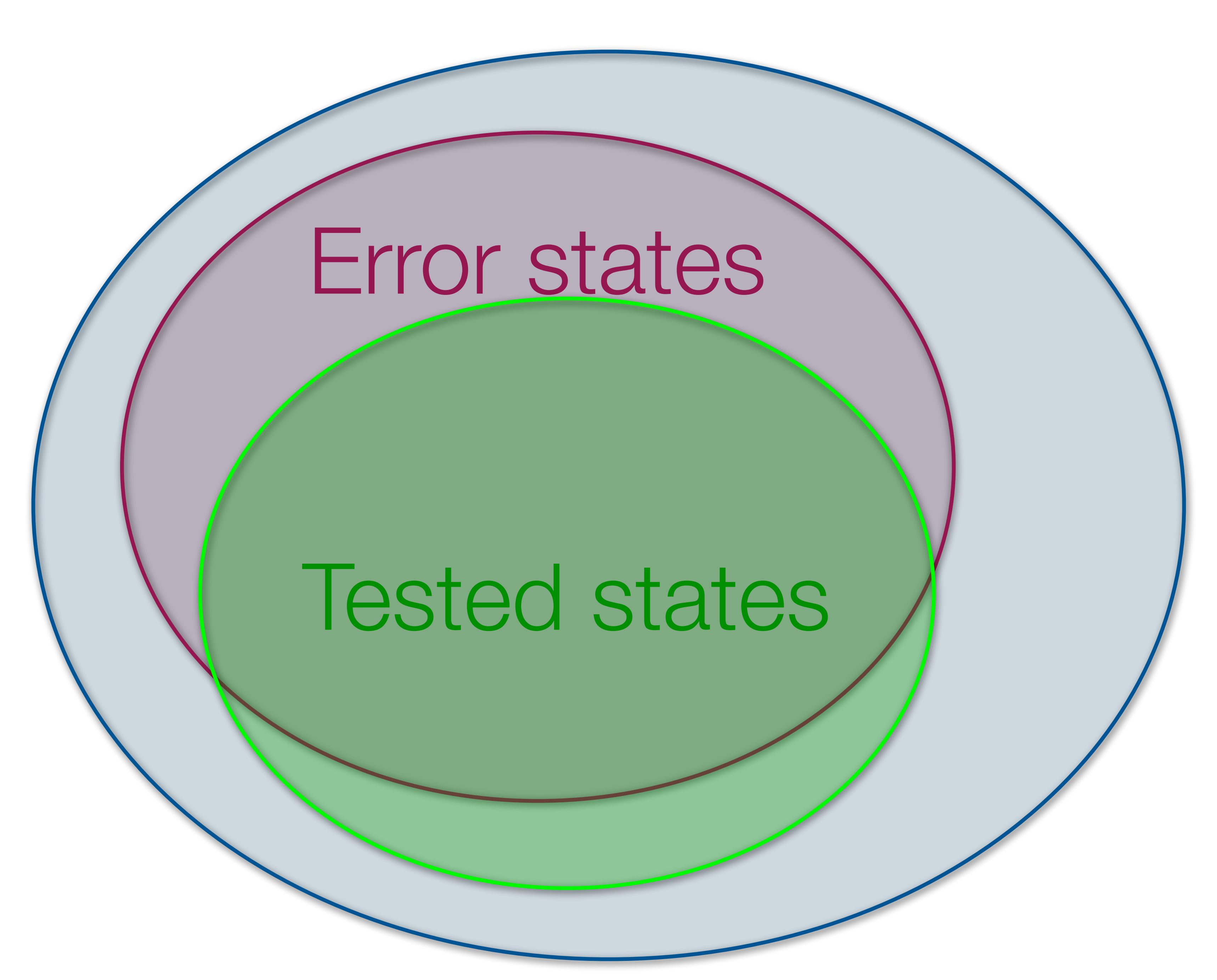
Possible states of your application



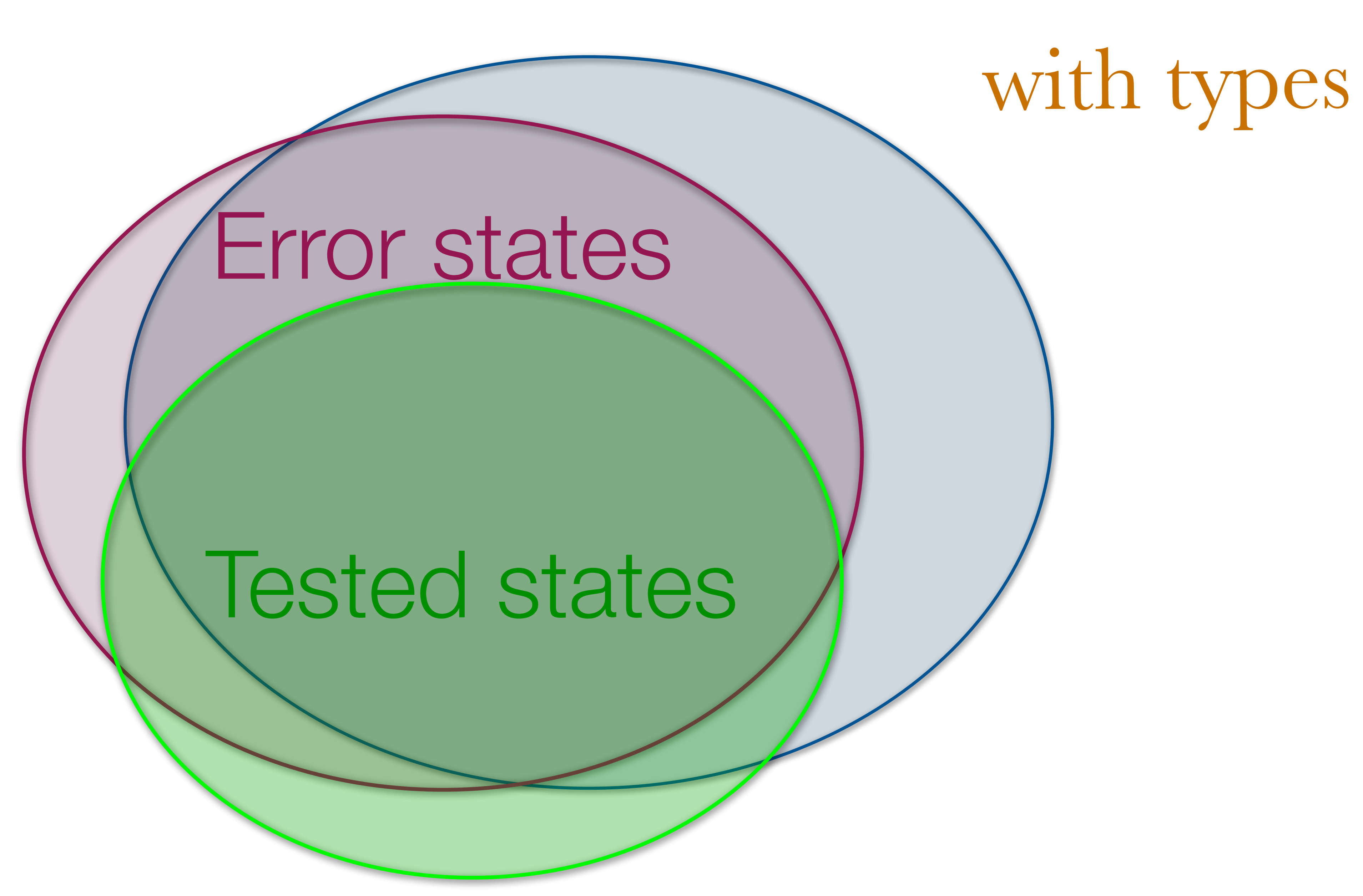
Possible states of your application



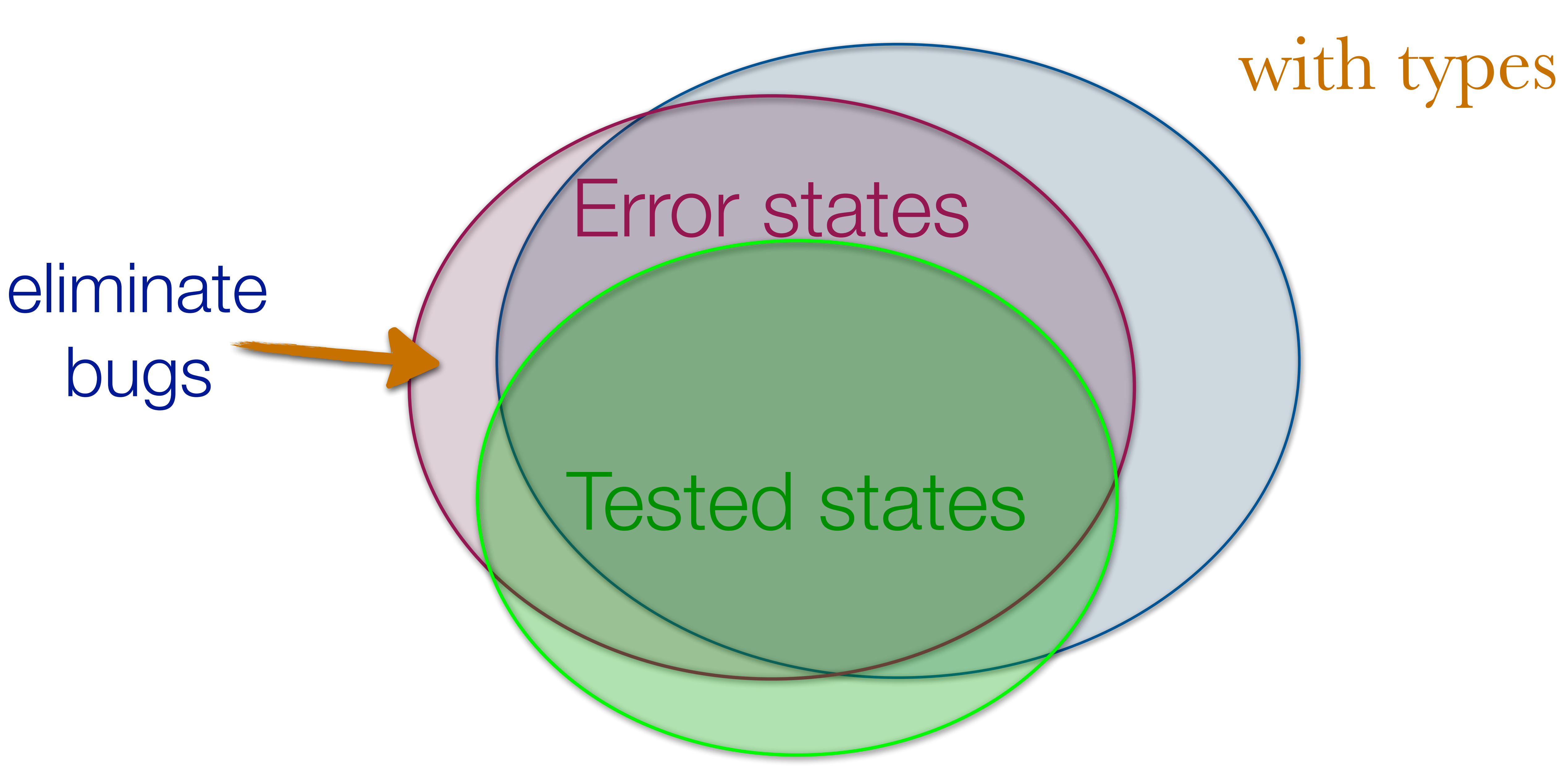
Possible states of your application



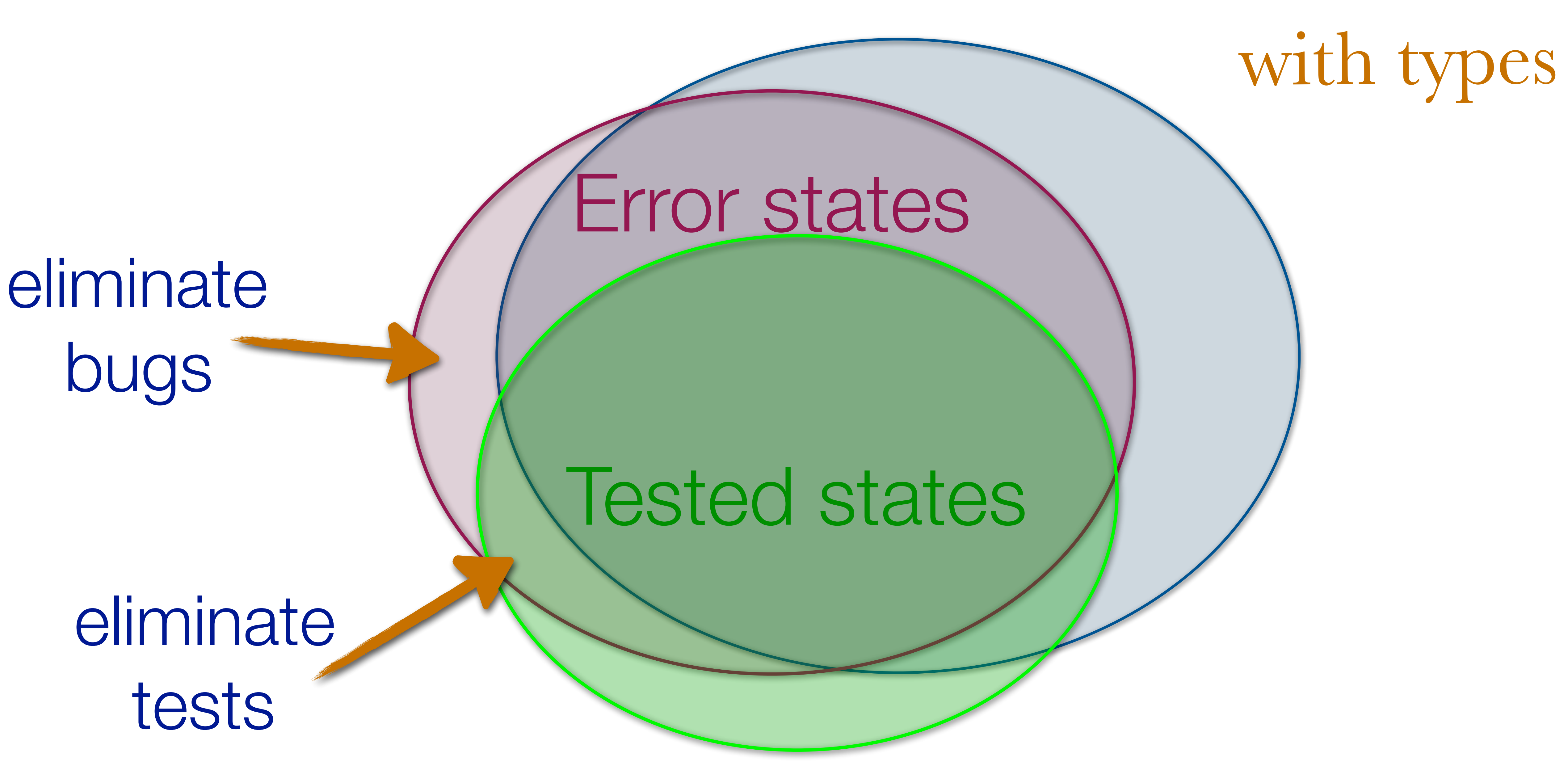
Possible states of your application



Possible states of your application

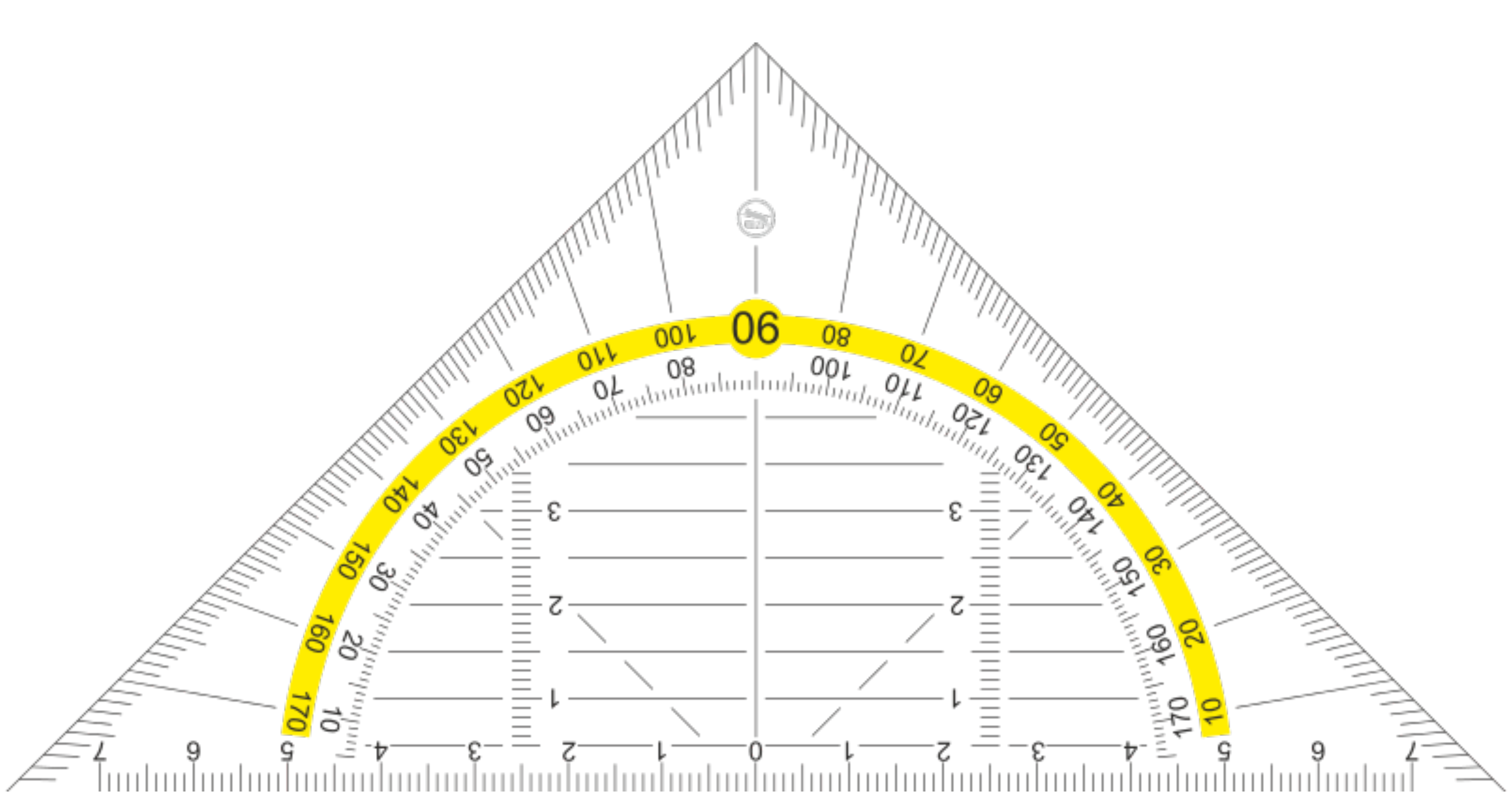


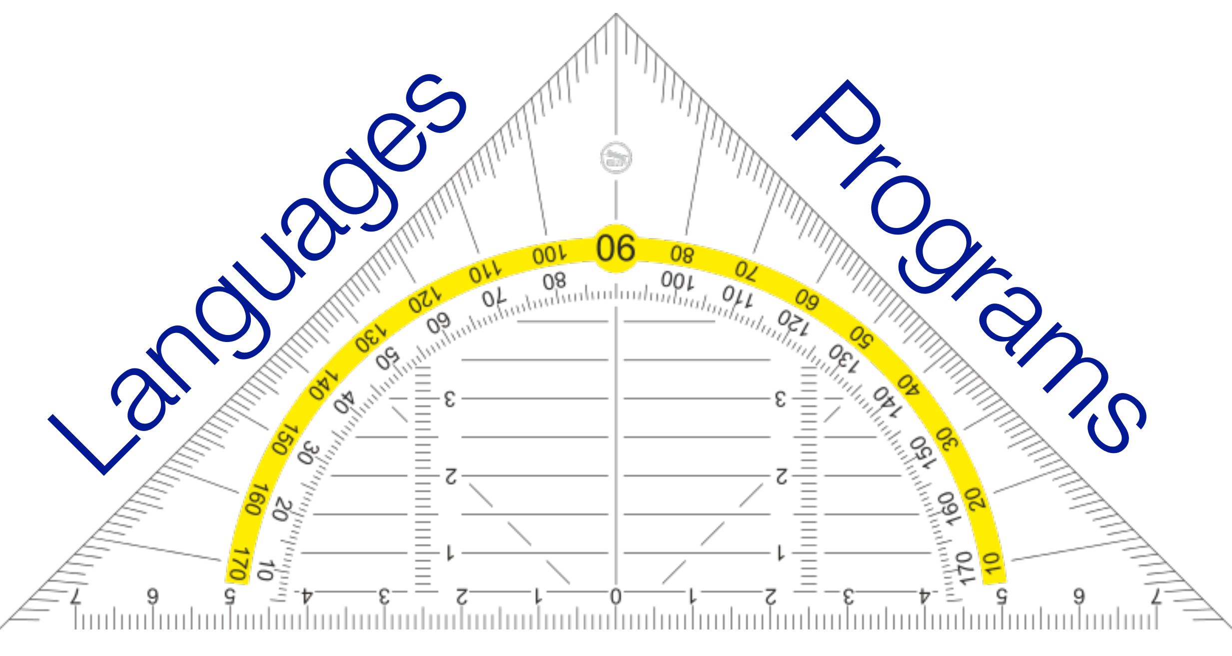
Possible states of your application



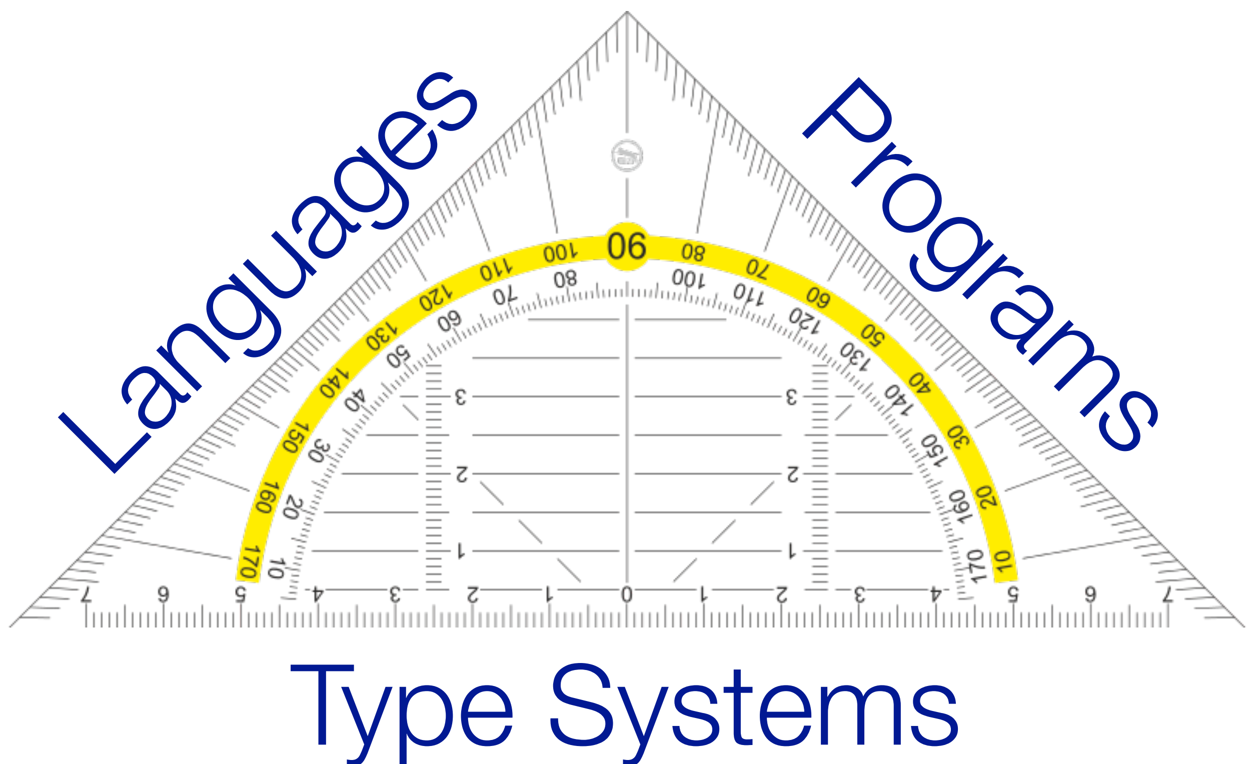
Are types restraints?

Types are a design tool!





Type Systems



“Make undesirable states unrepresentable.”

"Swift encourages typed functional programming in a mainstream language."

"Swift encourages typed functional programming in a mainstream language."

“Swift encourages typed functional programming in a mainstream language.”

Value types

Immutable model & UI state machines

"Swift encourages typed functional programming in a mainstream language."

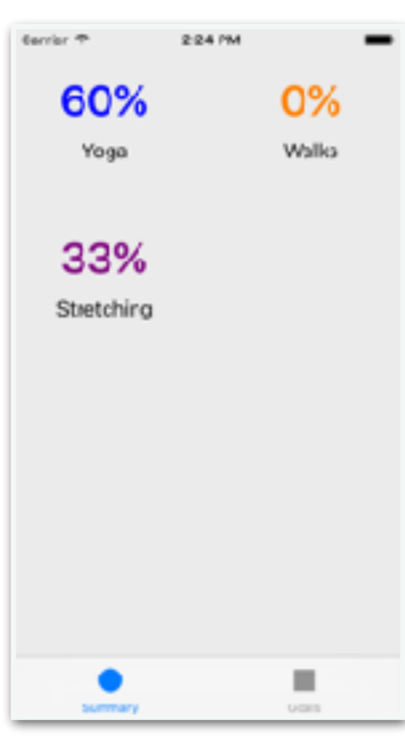
Value types

Immutable model & UI state machines

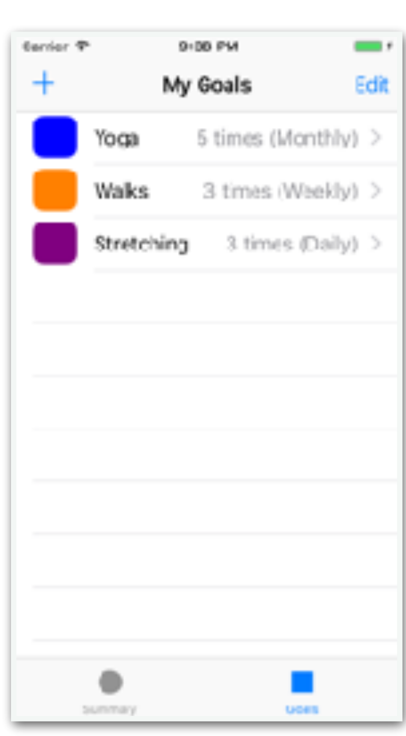
Protocols with associated types

Generic change propagation

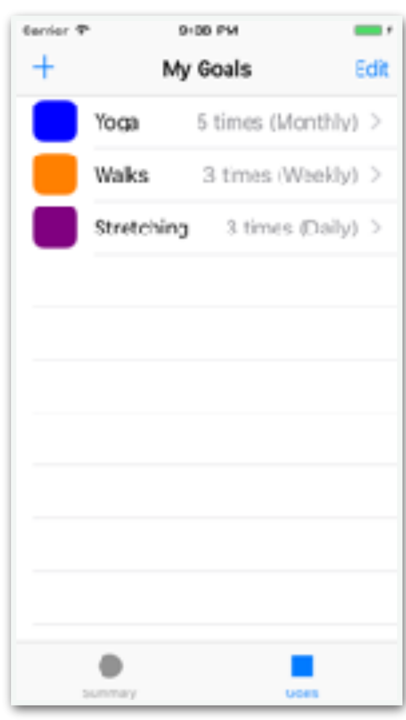
Goals.app

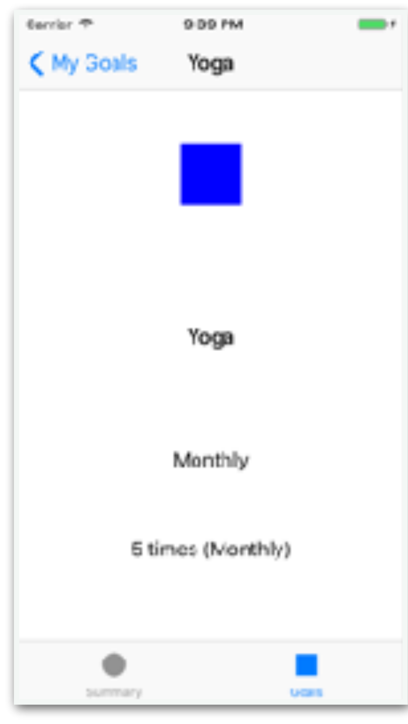


Goals.app

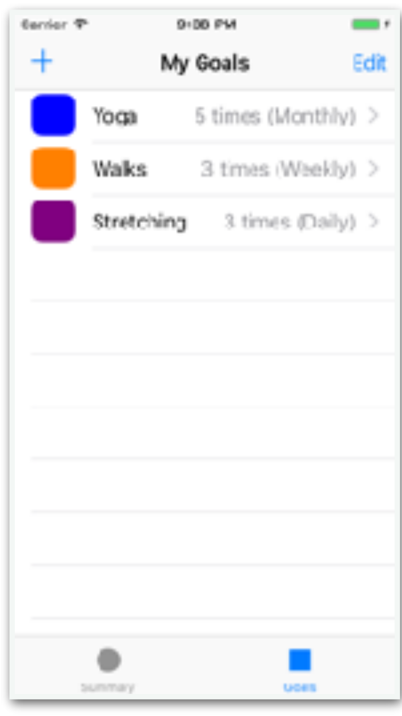


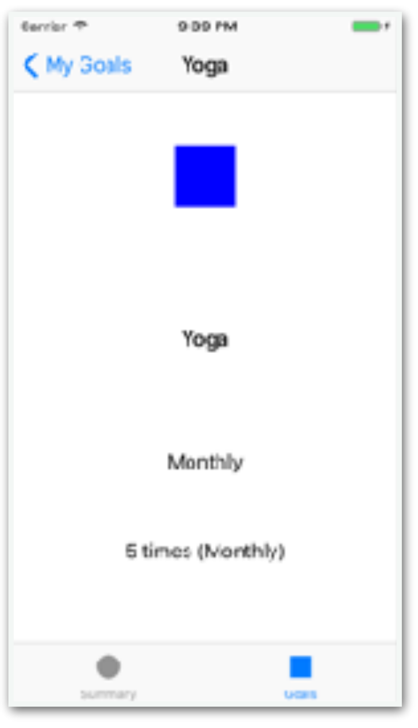
Goals.app

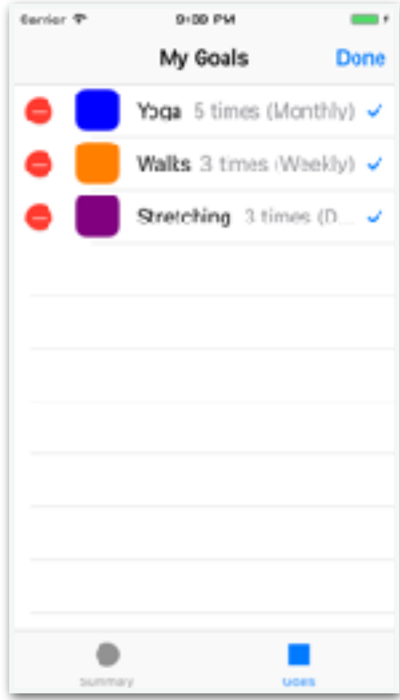




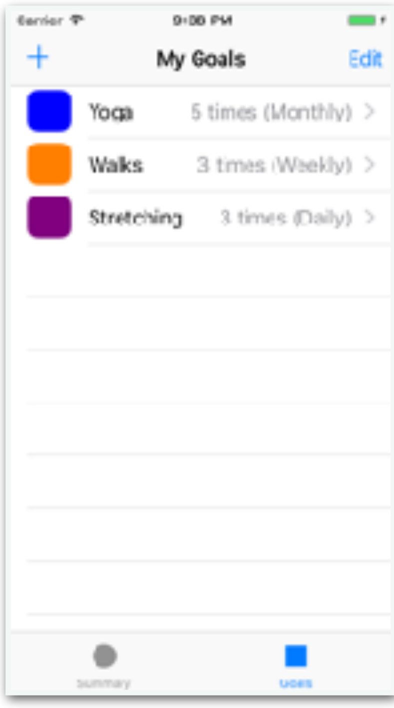
Goals.app

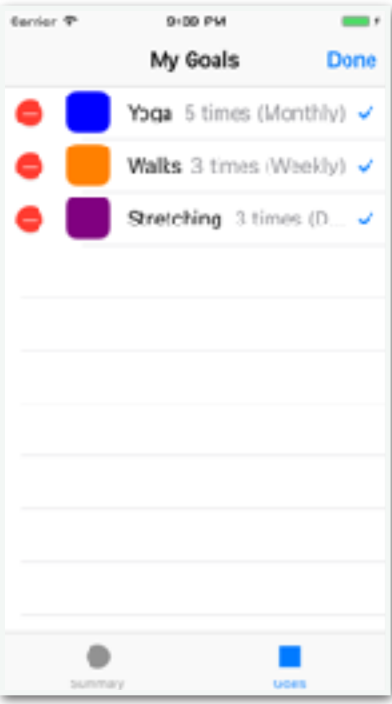


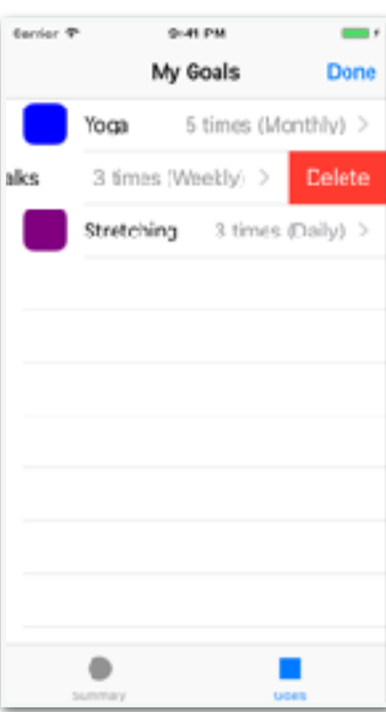




Goals.app







Value Types
Localise & structure change
CROWNE MINT

Value types in Swift

Value types in Swift

struct

```
struct Point {
  var x: Double
  var y: Double
}
```

Value types in Swift

struct

```
struct Point {
  var x: Double
  var y: Double
}
```

enum with associated values

```
enum Result<ResultType> {
  case result(value: ResultType)
  case error(err: NSError)
}
```

Value types in Swift

struct

```
struct Point {
  var x: Double
  var y: Double
}
```

generic type parameter

enum with associated values

```
enum Result<ResultType> {
  case result(value: ResultType)
  case error(err: NSError)
}
```

Kolin Edition

```
// Data class (similar to struct)
data class Point(var x: Double, var y: Double)

// Sealed class (similar to enum)
sealed class Result {
  data class Success<T>(val value: T)  : Result()
  data class Error(val err: Exception) : Result()
}
```

Reference types versus value types

```
var v: RefPnt

var w: RefPnt
```

```
class RefPnt { var x, y: Double … }
```

Reference types versus value types

```
var v: RefPnt
```

```
x: 0, y: 0
```

```
var w: RefPnt
```

```
class RefPnt { var x, y: Double … }

v = RefPnt(x: 0, y: 0)
```

Reference types versus value types

```
var v: RefPnt
var w: RefPnt
```

```
x: 0, y: 0
```

```
class RefPnt { var x, y: Double … }

v = RefPnt(x: 0, y: 0)
w = v
```

Reference types versus value types

```
var v: RefPnt
var w: RefPnt
```

x: 10, y: 0

```
class RefPnt { var x, y: Double … }

v = RefPnt(x: 0, y: 0)
w = v
w.x = 10
```

Reference types versus value types

```
var v: Point

var w: Point
```

```
struct Point { var x, y: Double }
```

Reference types versus value types

```
var v: Point ⟶ x: 0, y: 0

var w: Point
```

```
struct Point { var x, y: Double }

v = Point(x: 0, y: 0)
```

Reference types versus value types

var v: Point → x: 0, y: 0

var w: Point → x: 0, y: 0

```
struct Point { var x, y: Double }

v = Point(x: 0, y: 0)
w = v
```

Reference types versus value types

```
var v: Point ⟶ x: 0, y: 0

var w: Point ⟶ x: 10, y: 0
```

```
struct Point { var x, y: Double }

v = Point(x: 0, y: 0)
w = v
w.x = 10
```

Value types

```
var v: Point  →  x: 0, y: 0

var w: Point  →  x: 10, y: 0
```

Value types

```
var v: Point → x: 0, y: 0
var w: Point → x: 10, y: 0
```

Localise change

Value types

```
var v: Point → x: 0, y: 0

var w: Point → x: 10, y: 0
```

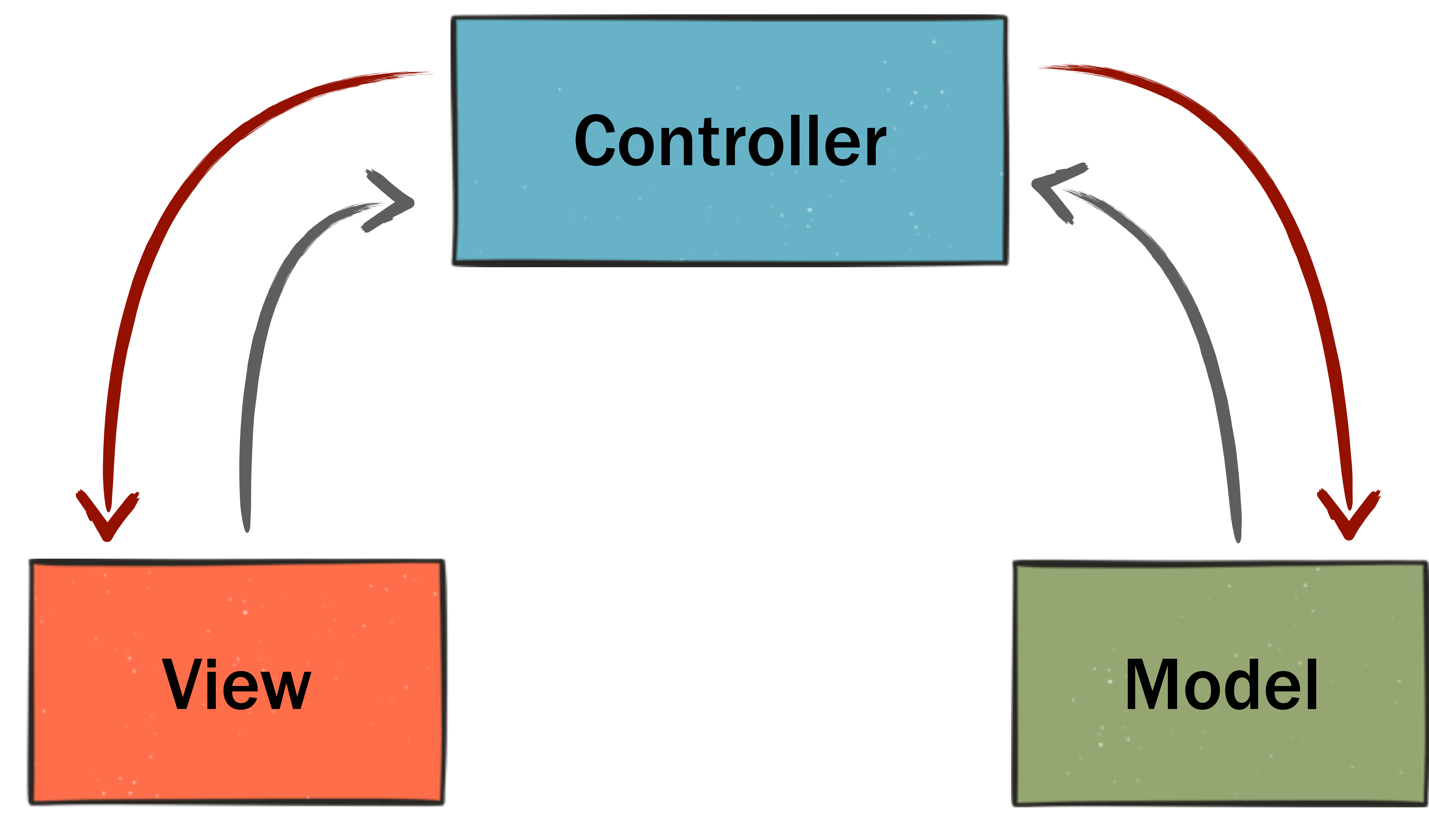
Localise change

Facilitate local reasoning

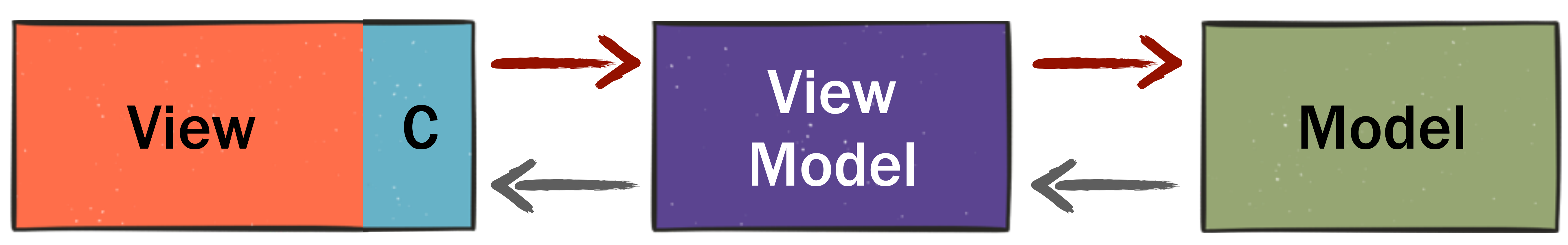
Value Types

Immutable Model

MVC & MVVM Architectures

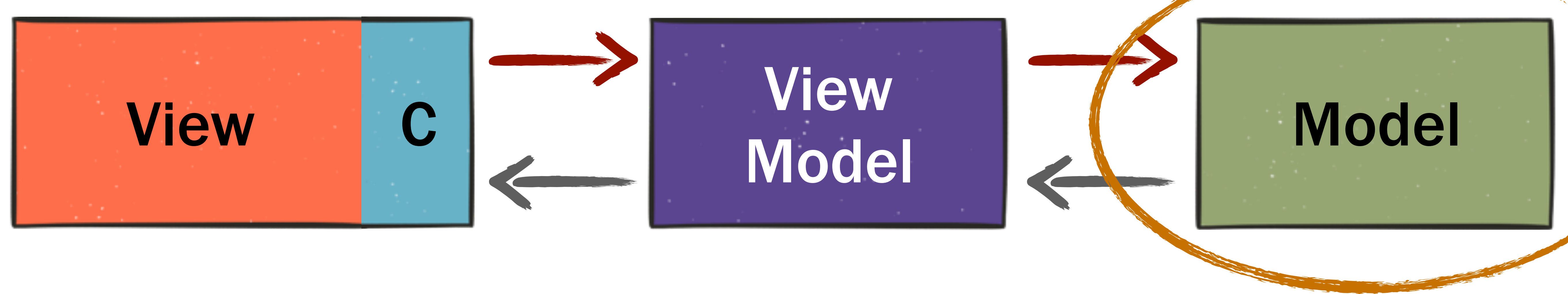


MVC & MVVM Architectures



MVC & MVVM Architectures

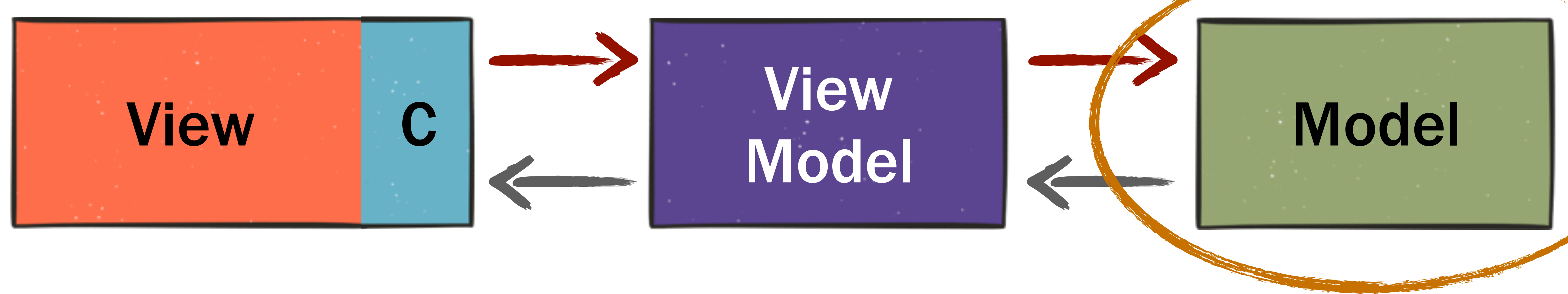
passed around as
reference type



MVC & MVVM Architectures

× Accidental changes

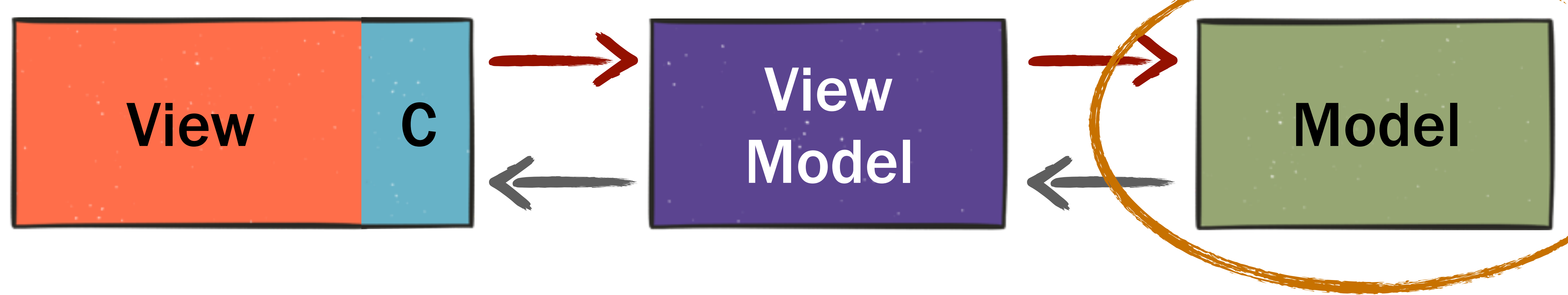
passed around as
reference type



MVC & MVVM Architectures

- × Accidental changes
- × Changes in wrong order

passed around as
reference type



Immutable Goals

```
struct Goal {
  let uuid:      UUID                    // fast equality
  var colour:    UIColor
  var title:     String
  var interval:  GoalInterval
  var frequency: Int
}
```

Immutable Goals

```
struct Goal {
  let uuid:      UUID                  // fast equality
  var colour:    UIColor
  var title:     String
  var interval:  GoalInterval
  var frequency: Int
}

typealias GoalProgress = (goal: Goal, count: Int?)
```

Immutable Goals

```
struct Goal {
  let uuid:       UUID                    // fast equality
  var colour:     UIColor
  var title:      String
  var interval:   GoalInterval
  var frequency: Int
}

typealias GoalProgress = (goal: Goal, count: Int?)

typealias Goals = [GoalProgress]  // array
```

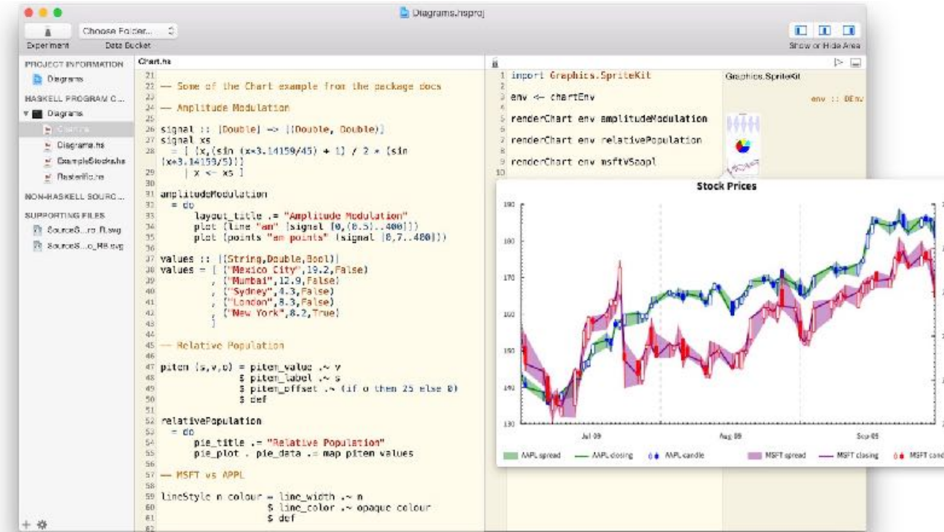
Kolin Edition

```kotlin
data class Goal(
        val uuid: UUID,
        var colour: Color,
        var title: String,
        var interval: GoalInterval,
        var frequency: Int)

typealias GoalProgress = Pair<Goal, Int?>

typealias Goals = Array<GoalProgress>
```

Immutable Models Do Scale!

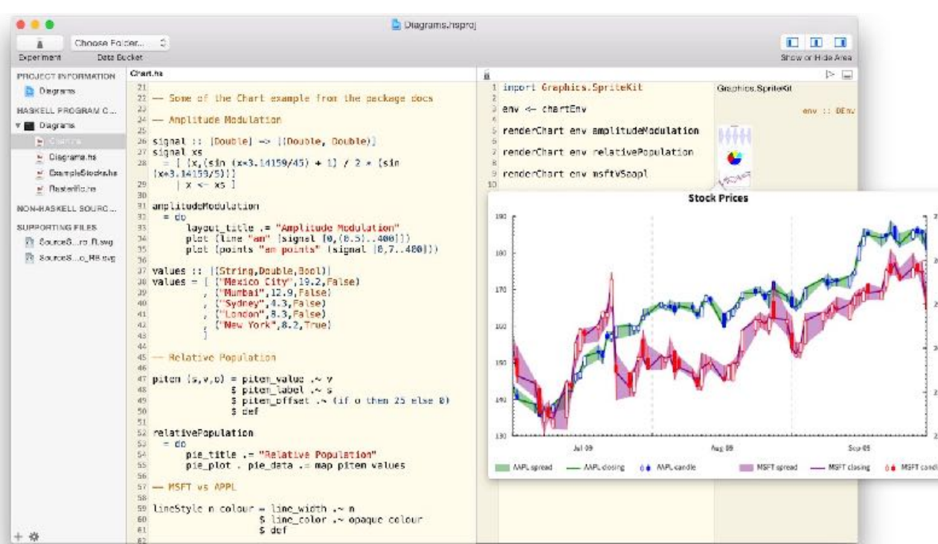


Immutable Models Do Scale!

Haskell for Mac

Uses an immutable model

Adopts Cocoa Document Architecture



Immutable Models Do Scale!

Haskell for Mac

Uses an immutable model

Adopts Cocoa Document Architecture

"Functional Programming in a Stateful World"
YOW! Lambda Jam 2015
(on speakerdeck.com)

But what if…

But what if…

"Use an immutable wrapper API to wrap mutable structures."

But what if…

"Use an immutable wrapper API to wrap mutable structures."

✓ React (Native)

Virtual DOM wraps DOM

But what if…

"Use an immutable wrapper API to wrap mutable structures."

- ✓ React (Native)
  Virtual DOM wraps DOM
- ✓ Haskell SpriteKit — Haskell library binding
  Algebraic data type wraps SpriteKit's mutable object graph

Replacing Tests

Replacing Tests

✓ Avoid accidental changes of model state

Replaces integration tests

Replacing Tests

✓ Avoid accidental changes of model state

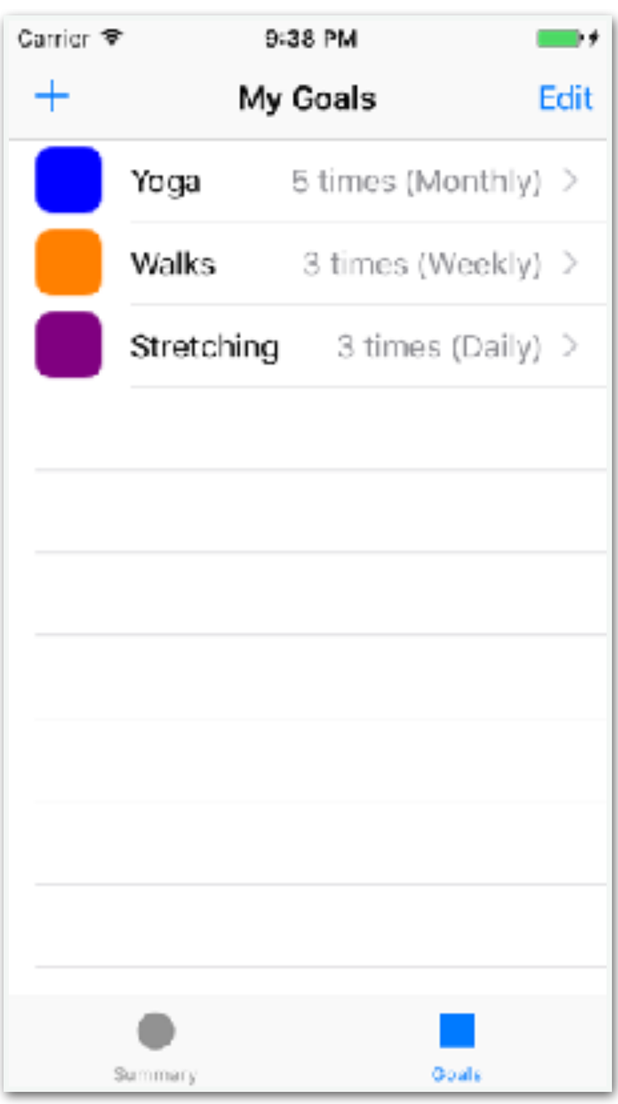
Replaces integration tests

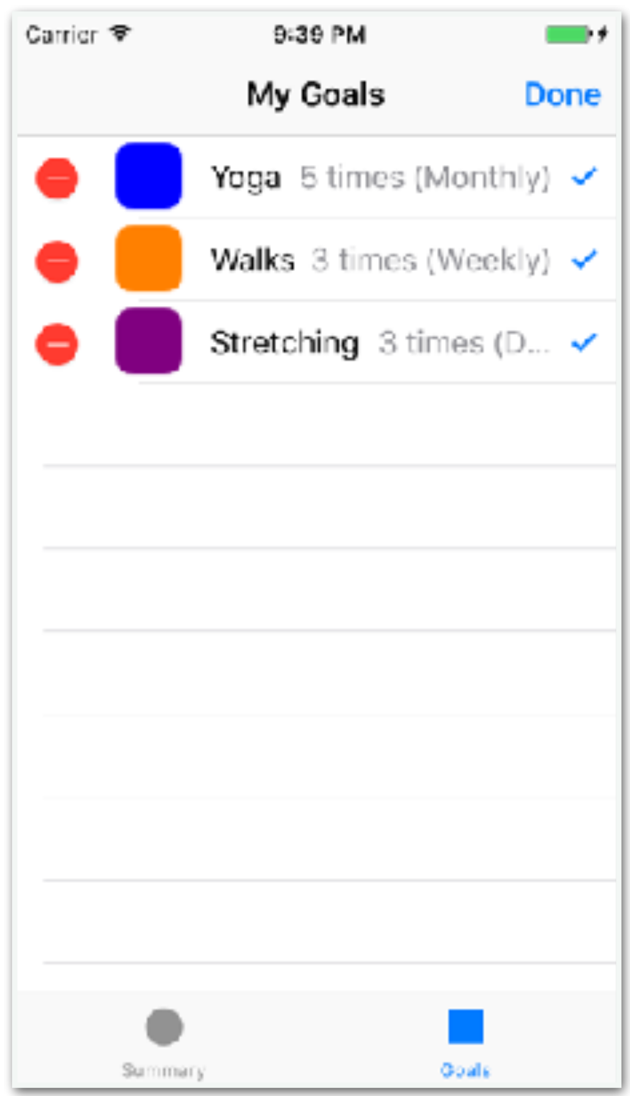
✓ Avoid races on concurrent threads

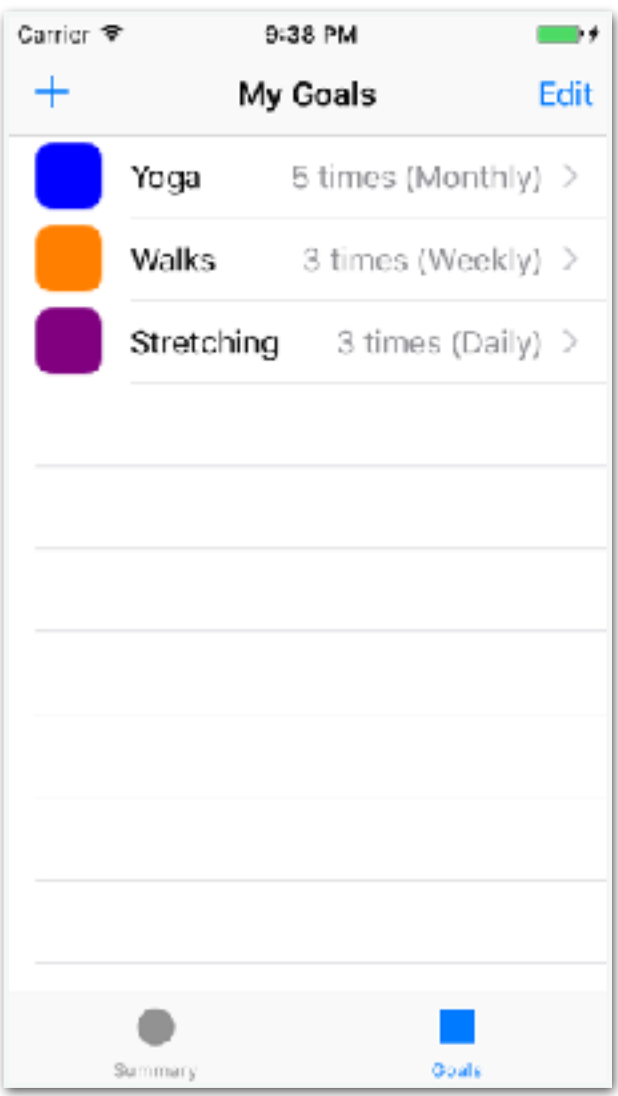
Often not properly tested

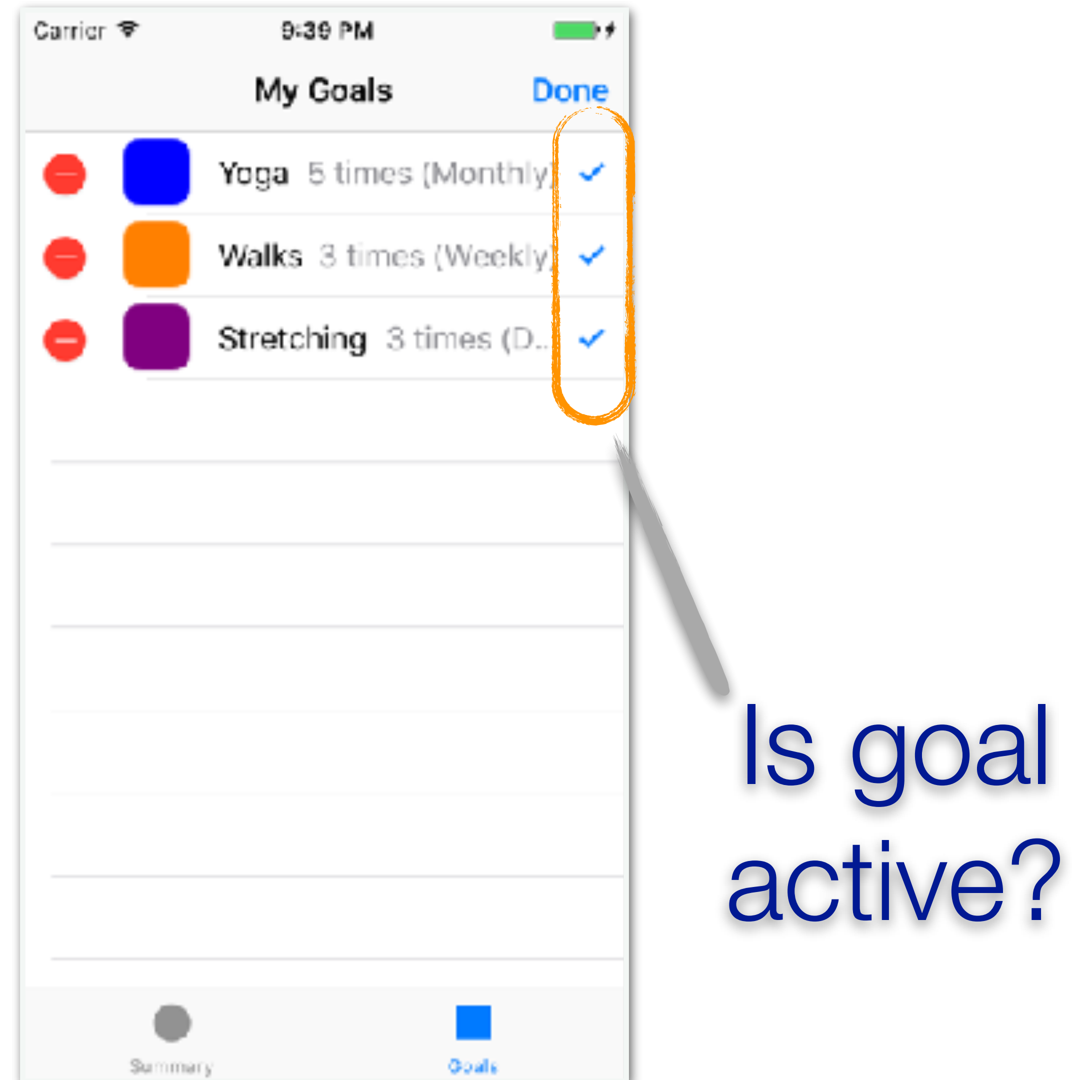
Enums with Associated Values

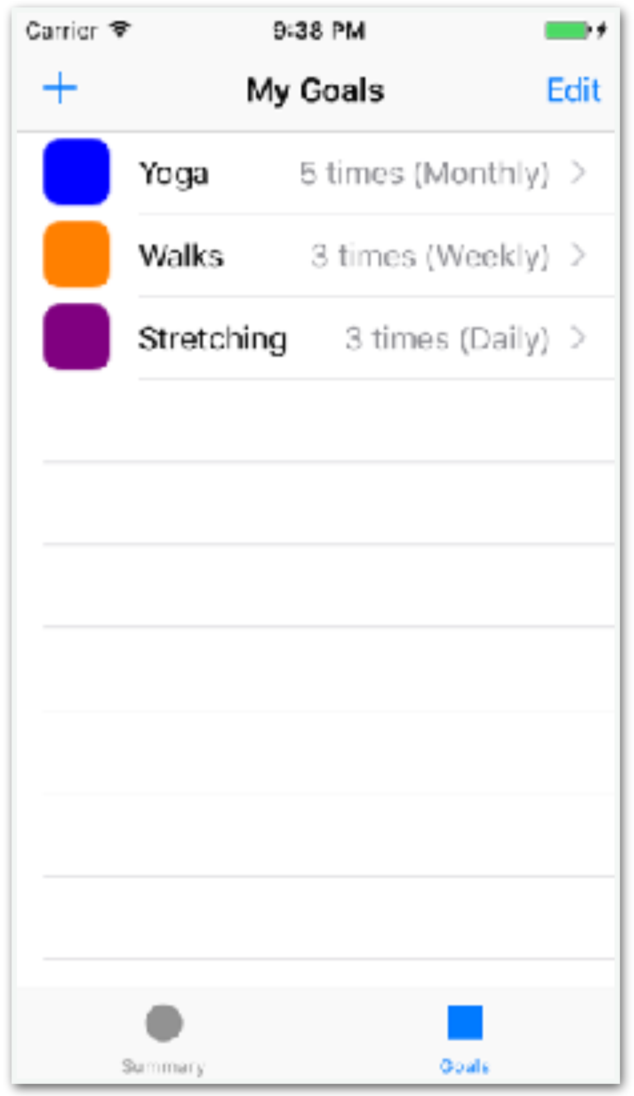
UI State Machines

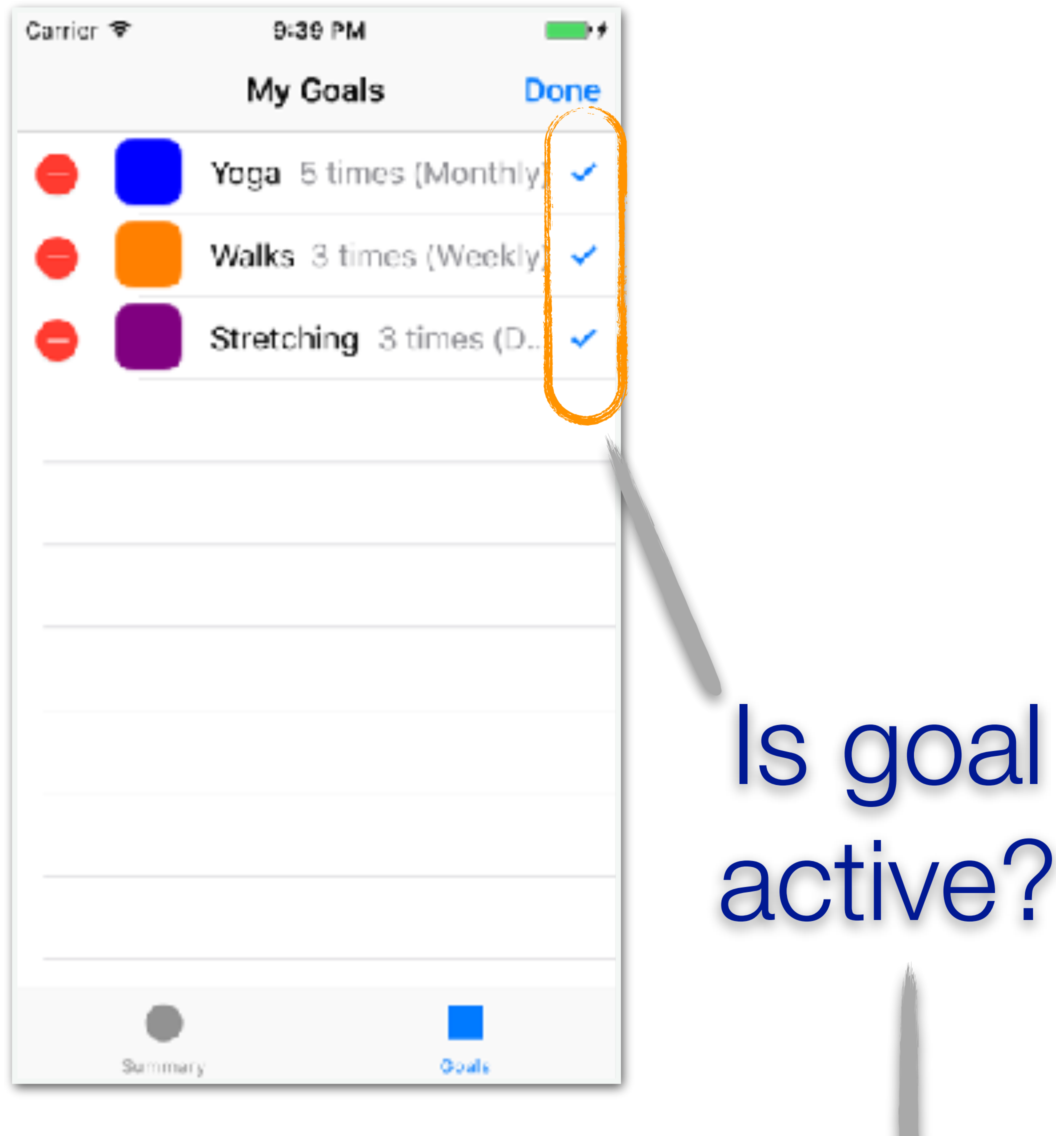

Carrier
9:38 PM
My Goals
Edit
Yoga
5 times (Monthly)
Walks
3 times (Weekly)
Stretching
3 times (Daily)


Carrier
9:39 PM
My Goals
Done
Yoga
5 times (Monthly)
Walks
3 times (Weekly)
Stretching
3 times (D...

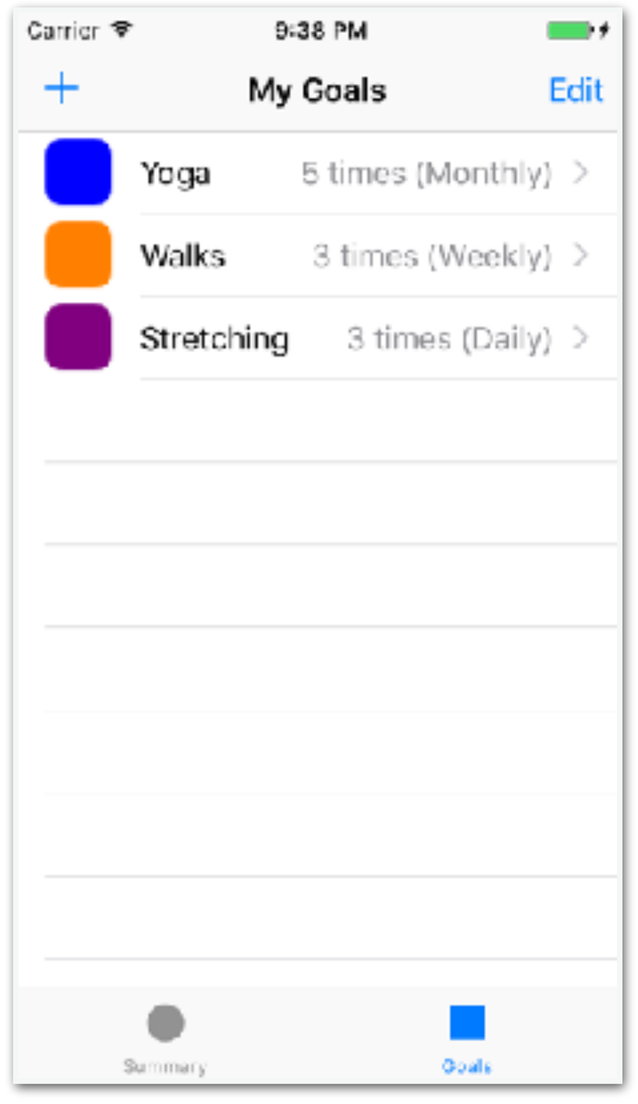

Carrier
9:38 PM
+
My Goals
Edit
Yoga
5 times (Monthly)
Walks
3 times (Weekly)
Stretching
3 times (Daily)

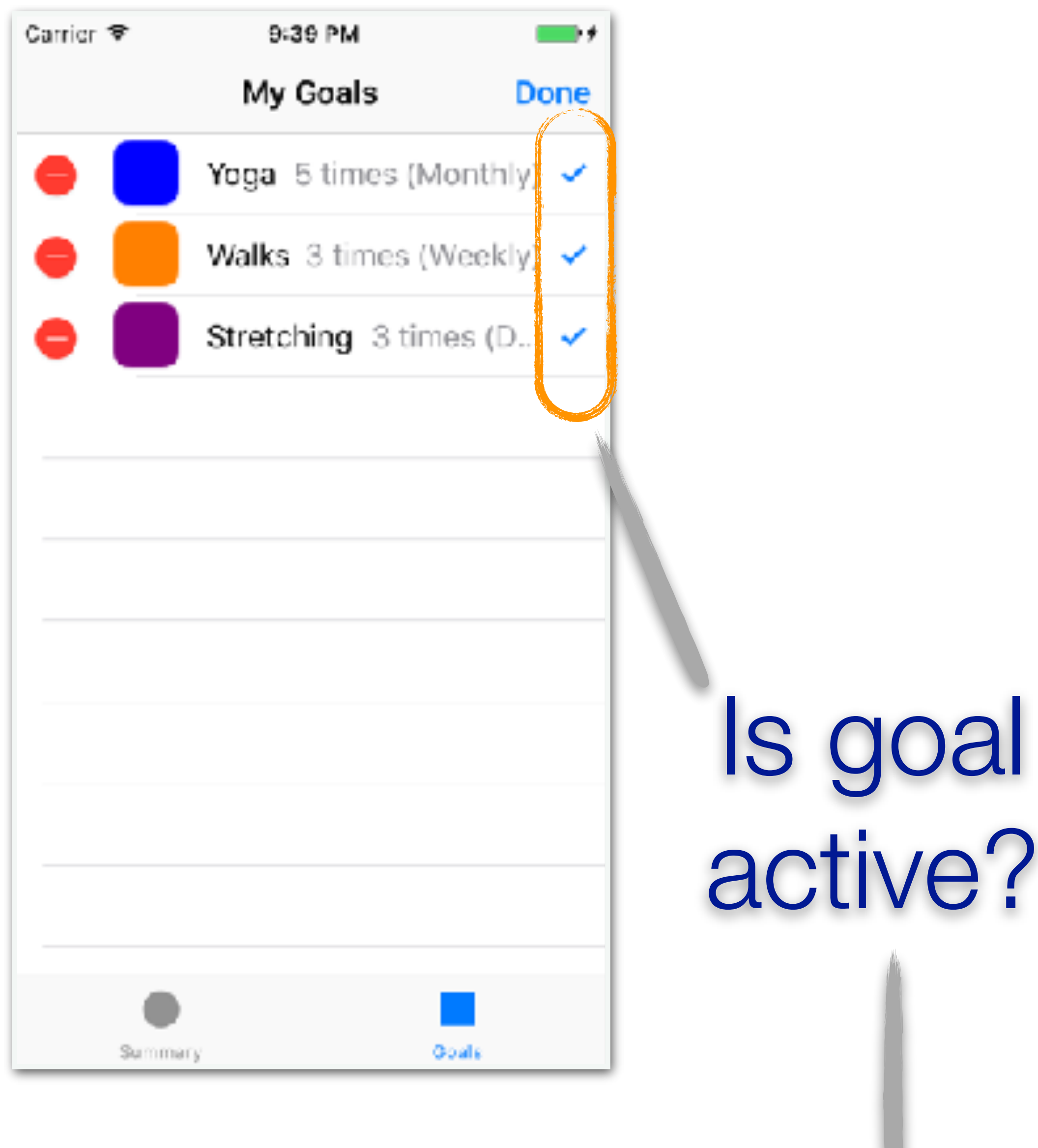

Carrier
9:39 PM
My Goals
Done
Yoga
5 times (Monthly)
Walks
3 times (Weekly)
Stretching
3 times (D...
Is goal active?





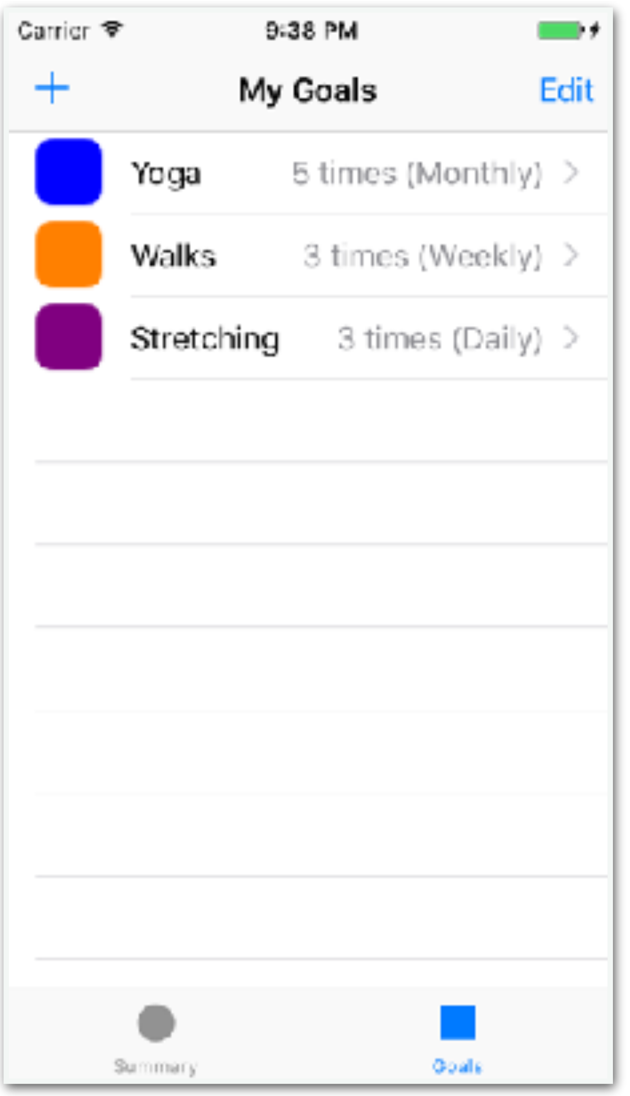

```
typealias GoalProgress = (goal: Goal, count: Int?)
typealias Goals = [GoalProgress]
```

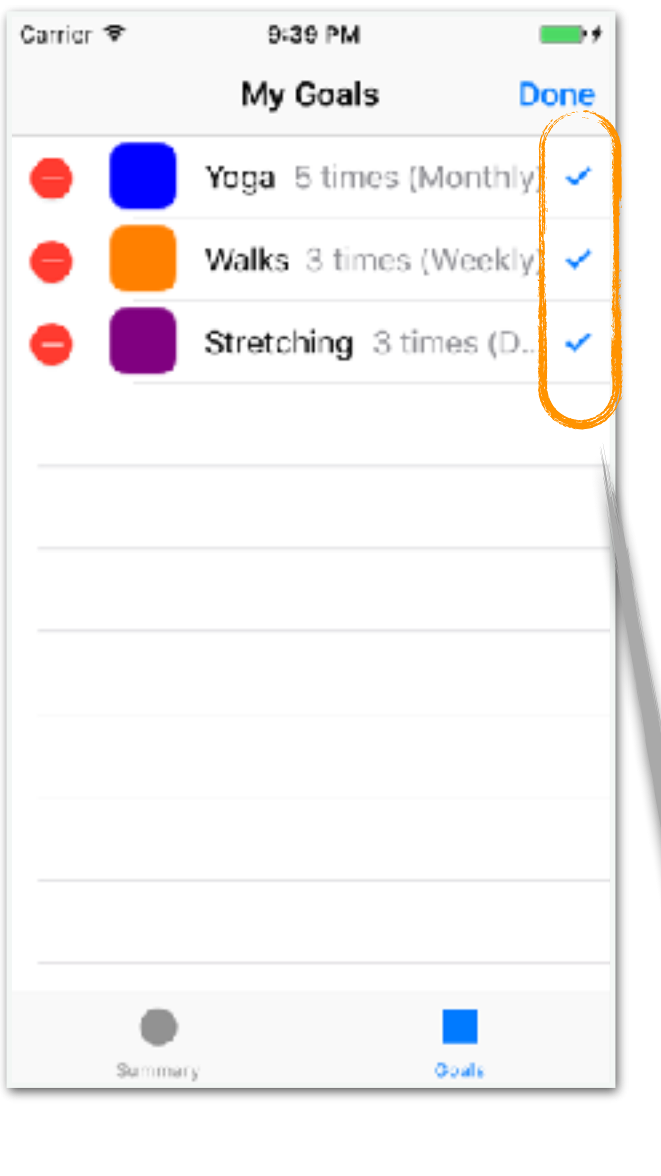




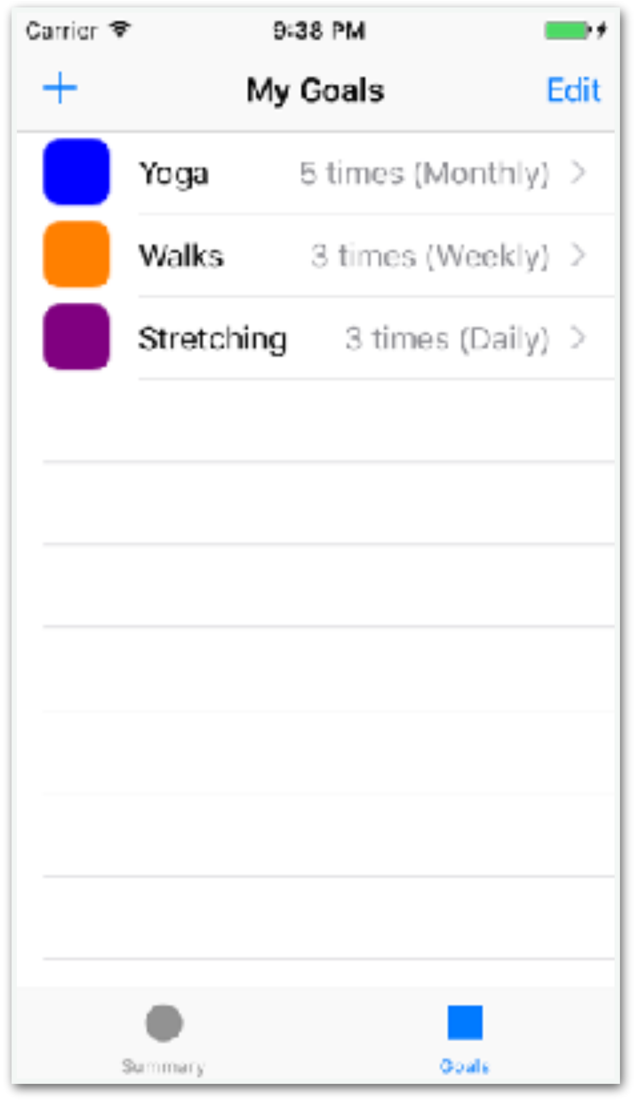

```
typealias GoalProgress = (goal: Goal, count: Int?)
typealias Goals = [GoalProgress]
```

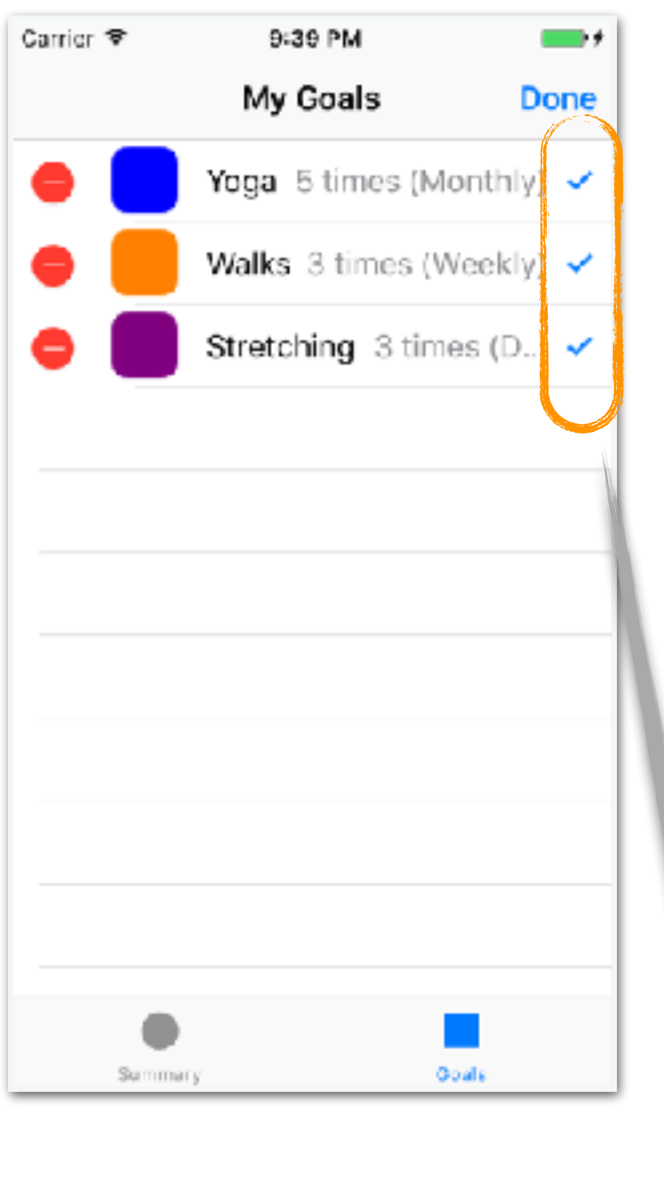
Need to be careful not to lose progress info






```
var goalsActivity: [Bool]
```






```
var goalsActivity: [Bool]
```

Not valid outside of editing mode

```
enum GoalsEditState {
  case displaying
  case editing
}
```

```
var goalsActivity: [Bool]
```

```swift
enum GoalsEditState {
  case displaying
  case editing(goalsActivity: [Bool])
}
```

```
enum GoalsEditState {
  case displaying
  case editing(goalsActivity: [Bool])
}
```

GoalsController: UITableViewController

```
var editState: GoalsEditState = .displaying
…
```

```
enum GoalsEditState {
  case displaying
  case editing(goalsActivity: [Bool])
}
```

GoalsController: UITableViewController

```
var editState: GoalsEditState = .displaying
…
switch editState {          // change mode
```

```swift
enum GoalsEditState {
  case displaying
  case editing(goalsActivity: [Bool])
}
```

GoalsController: UITableViewController

```swift
var editState: GoalsEditState = .displaying
…
switch editState {          // change mode
 case .displaying:
   navigationItem.setLeftBarButton(nil, …)
   editState = .editing(goalsActivity:
                           dataSource.goalsActivity)
```

alter
UI state

```swift
enum GoalsEditState {
  case displaying
  case editing(goalsActivity: [Bool])
}
```

GoalsController: UITableViewController

```swift
var editState: GoalsEditState = .displaying
…
switch editState {        // change mode
 case .displaying:
   navigationItem.setLeftBarButton(nil, …)
   editState = .editing(goalsActivity:
                         dataSource.goalsActivity)

 case .editing(let goalsActivity):
   navigationItem.setLeftBarButton(addButton, …)
   dataSource.commitGoalsActivity(goalsActivity)
   editState = .displaying }
```

alter
UI state

```swift
enum GoalsEditState {
  case displaying
  case editing(goalsActivity: [Bool])
}
```

GoalsController

```swift
guard case .editing(var goalsActivity) = editState
  else { return }
```

```
enum GoalsEditState {
  case displaying
  case editing(goalsActivity: [Bool])
}
```

access to mode-specific state is guarded

GoalsController

```
guard case .editing(var goalsActivity) = editState
  else { return }
```

```
enum GoalsEditState {
  case displaying
  case editing(goalsActivity: [Bool])
}
```

access to mode-specific state is guarded

GoalsController

```
guard case .editing(var goalsActivity) = editState
  else { return }

let newIsActive = !goalsActivity[indexPath.item]
cell.editingAccessoryType = newIsActive ? .checkmark
                                        : .none
```

Kolin Edition

```
sealed class GoalsEditState {
 data class Displaying : GoalsEditState()
 data class Editing(val goalsActivity: Array<Boolean>)
    : GoalsEditState()
}
```

```
sealed class GoalsEditState {
  data class Displaying : GoalsEditState()
  data class Editing(val goalsActivity: Array<Boolean>)
     : GoalsEditState()
}
```

```
var editState: GoalsEditState
    = GoalsEditState.Displaying()
...
 when(editState) {
  is GoalsEditState.Displaying -> {
    navigationItem.setLeftBarButton(null, ...)
    editState = GoalsEditState.Editing(
              goalsActivity = dataSource.goalsActivity)
  }
  is GoalsEditState.Editing -> {
    navigationItem.setLeftBarButton(addButton, ...)
      dataSource.commitGoalsActivity(goalsActivity)
      editState = GoalsEditState.Displaying()
  }
}
```

Replacing Tests

Replacing Tests

✓ Avoid inconsistent state changes

Replaces UI tests

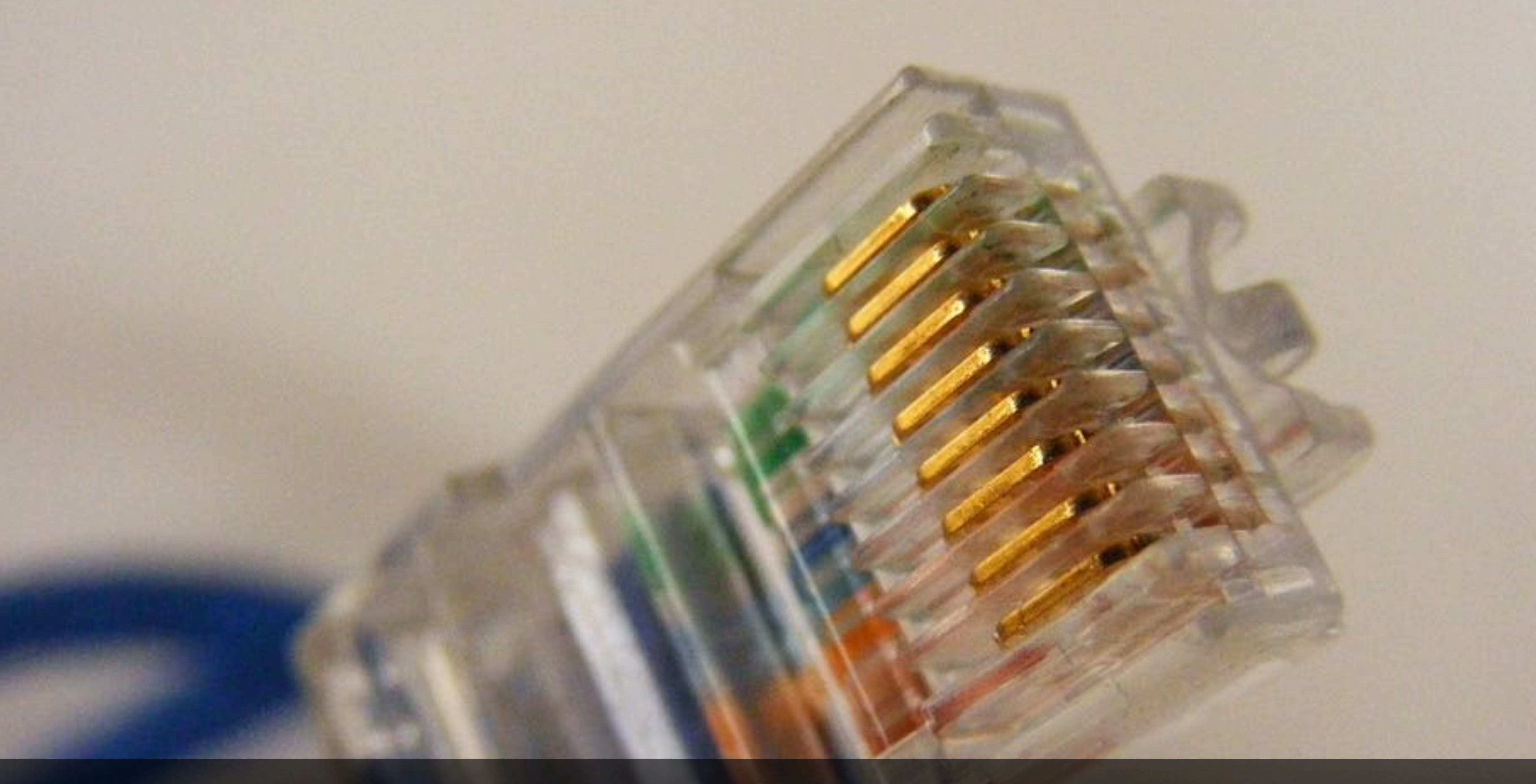
Replacing Tests

✓ Avoid inconsistent state changes

Replaces UI tests

✓ Avoid non-exhaustive transitions

Replaces exhaustiveness tests



Protocols with associated types

Structured change propagation

"Now that the model layer is immutable, how do we change anything?"

Δt: 0s

25

Mutable Variable

Δt: 1s

7

Mutable Variable

Δt: 2s

11

Mutable Variable

Δt: 3s

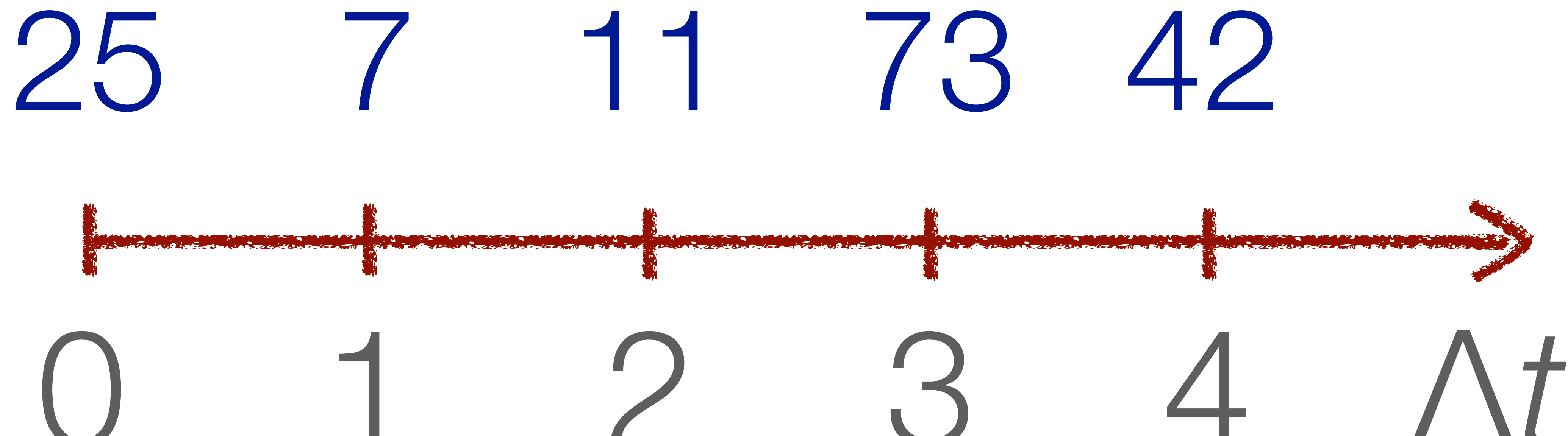
73

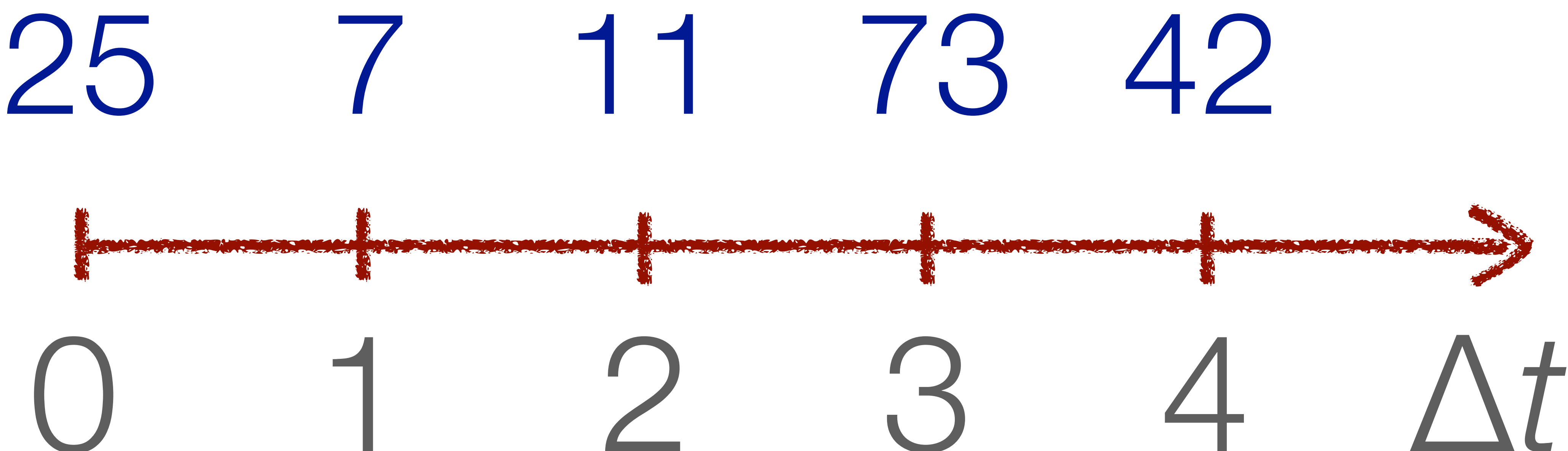
Mutable Variable

Δt: 4s

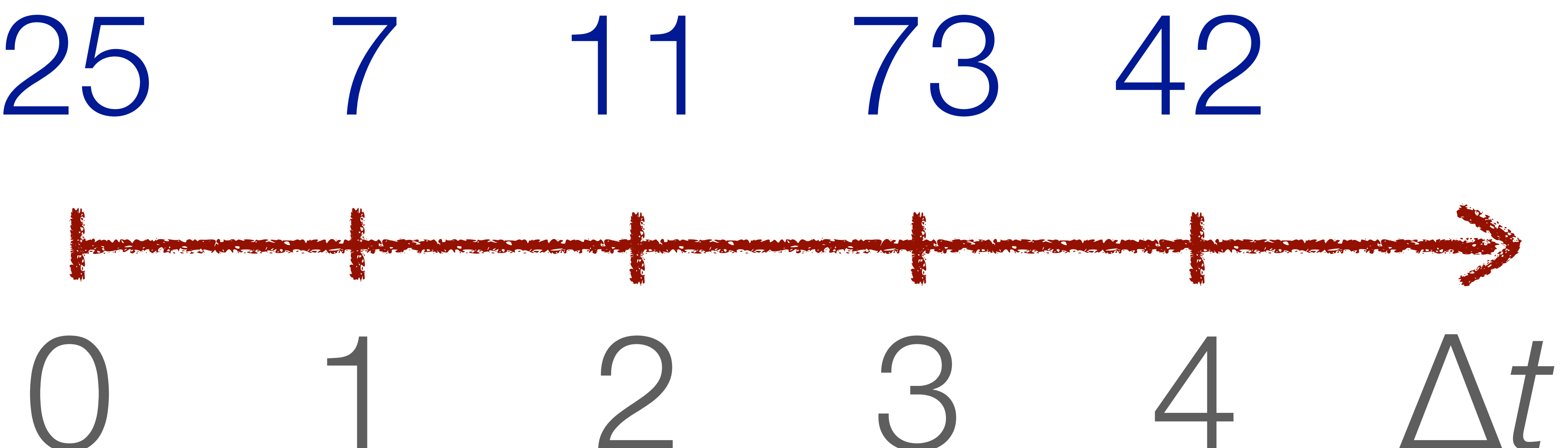
42

Mutable Variable


25
7
11
73
42
0
1
2
3
4
Δt



✓ Flow of changes explicit in code



✓ Flow of changes explicit in code

✓ Changes can be processed with combinators (map, fold, accumulate, and so on)

Changes API

```
protocol Observable: class {
 associatedtype ObservedValue
```

Changes API

```
protocol Observable: class {
 associatedtype ObservedValue

 func observe<Context: AnyObject>(
     withContext context: Context,
     observer: (Context, ObservedValue) -> ())
   -> ()
}
```

registered observer call back

Kolin Edition

```
interface Observable<ObservedValue> {
    fun <Context : Any> observe(
           context: Context,
           observer: (Context, ObservedValue) -> Unit)
}
```

Changes API

```
protocol Observable: class {
 associatedtype ObservedValue

 func observe<Context: AnyObject>(
     withContext context: Context,
     observer: (Context, ObservedValue) -> ())
   -> ()
}
```

registered observer
call back

```
protocol Observable: class {
 associatedtype ObservedValue

 func observe<Context: AnyObject>(
     withContext context: Context,
     observer: (Context, ObservedValue) -> ())
   -> ()
}
```

```
protocol Observable: class {
 associatedtype ObservedValue

 func observe<Context: AnyObject>(
     withContext context: Context,
     observer: (Context, ObservedValue) -> ())
   -> ()
}
```

Ephemeral stream of changes

```
class Changing<Value>: Observable {
  typealias ObservedValue = Value
  func announce(change: Value) { … }
  func observe … { … }
}
```

```
protocol Observable: class {
 associatedtype ObservedValue

 func observe<Context: AnyObject>(
     withContext context: Context,
     observer: (Context, ObservedValue) -> ())
   -> ()
}
```

Ephemeral stream of changes

```
class Changing<Value>: Observable {
  typealias ObservedValue = Value
  func announce(change: Value) { … }
  func observe … { … }
}
```

Add another change value into the stream

```
interface Observable<ObservedValue> {
    fun <Context : Any> observe(
           context: Context,
           observer: (Context, ObservedValue) -> Unit)
}
```

```
class Changing<ObservedValue>
       : Observable<ObservedValue> {
    fun announce(change: ObservedValue) { ... }
    fun observe ... { ... }
}
```

```
protocol Observable: class {
 associatedtype ObservedValue

 func observe<Context: AnyObject>(
     withContext context: Context,
     observer: (Context, ObservedValue) -> ())
   -> ()
}
```

Ephemeral stream of changes

```
class Changing<Value>: Observable {
  typealias ObservedValue = Value
  func announce(change: Value) { … }
  func observe … { … }
}
```

Add another change value into the stream

```
protocol Observable: class {
 associatedtype ObservedValue

 func observe<Context: AnyObject>(
     withContext context: Context,
     observer: (Context, ObservedValue) -> ())
   -> ()
}
```

Accumulating value

```
class Accumulating<Value, Accumulator>: Observable {
  typealias ObservedValue = Accumulator
  init <Observed: Observable where …> (
    observing observed: Observed,
    startingFrom initial: Accumulator,
    accumulate: (Value, Accumulator) -> Accumulator)
  func observe … { … } }
```

```
protocol Observable: class {
 associatedtype ObservedValue

 func observe<Context: AnyObject>(
     withContext context: Context,
     observer: (Context, ObservedValue) -> ())
   -> ()
}
```

Accumulating value

```
class Accumulating<Value, Accumulator>: Observable {
  typealias ObservedValue = Accumulator
  init <Observed: Observable where …> (
    observing observed: Observed,
    startingFrom initial: Accumulator,
    accumulate: (Value, Accumulator) -> Accumulator)
  func observe … { … } }
```

Accumulating change

```kotlin
interface Observable<ObservedValue> {
    fun <Context : Any> observe(
        context: Context,
        observer: (Context, ObservedValue) -> Unit)
}
```

```kotlin
class Accumulating<Value, Acc>(
  val observed: Observed,
  val initial: Acc,
  val accumulate: (Value, Acc) -> Acc)
    : Observable<Acc> {
  fun observe ... { ... }
}
```

```
extension Observable {

}
```

```
extension Observable {

 map     apply function to each observed value

}
```

```
extension Observable {

 map     apply function to each observed value

 merge   combine two streams of observations

}
```

```
extension Observable {

 map       apply function to each observed value

 merge     combine two streams of observations

 accumulate
           combine the values of a stream of
           observations using an accumulation
           function

 …

}
```

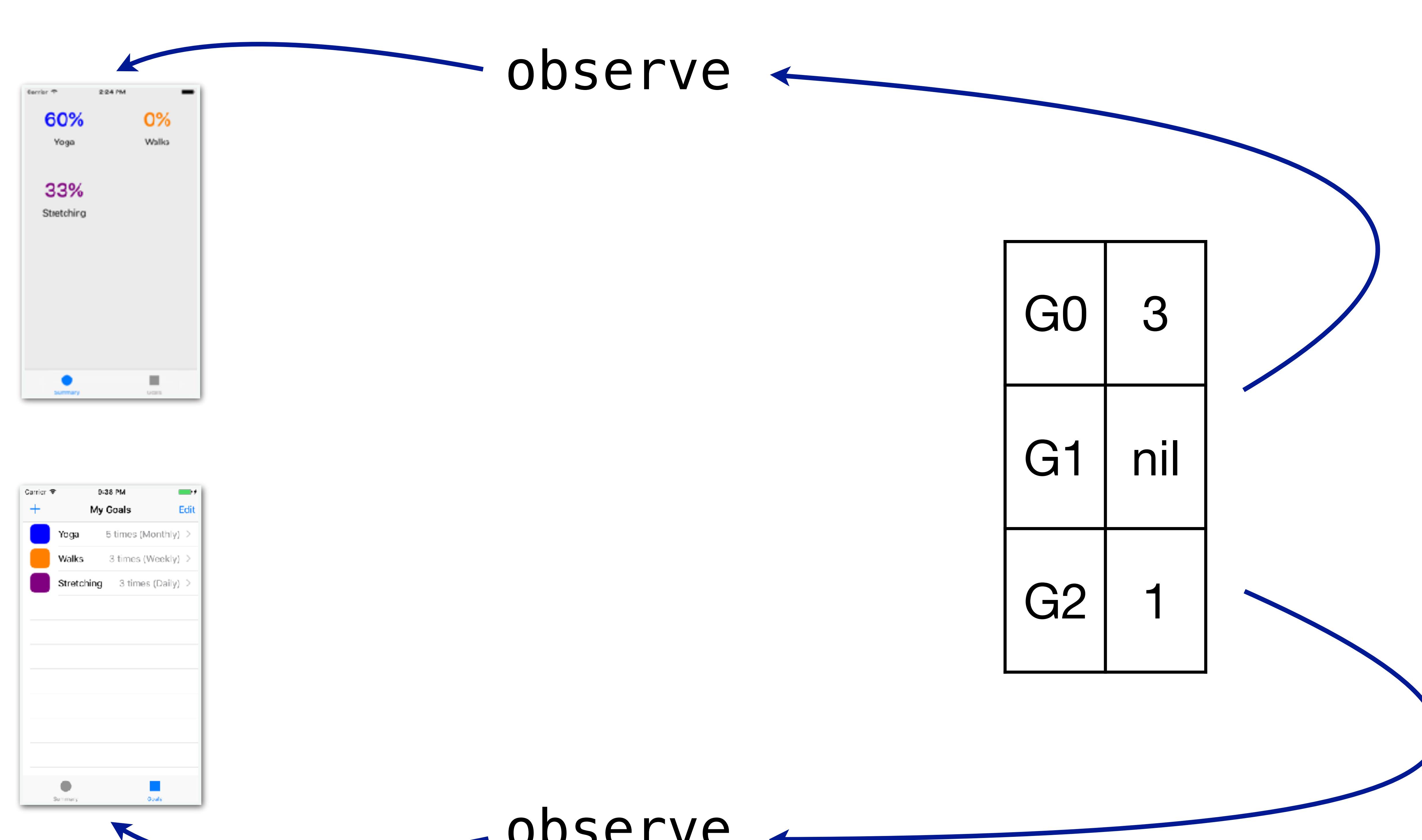
Change Propagation in Goals

G0	3
G1	nil
G2	1

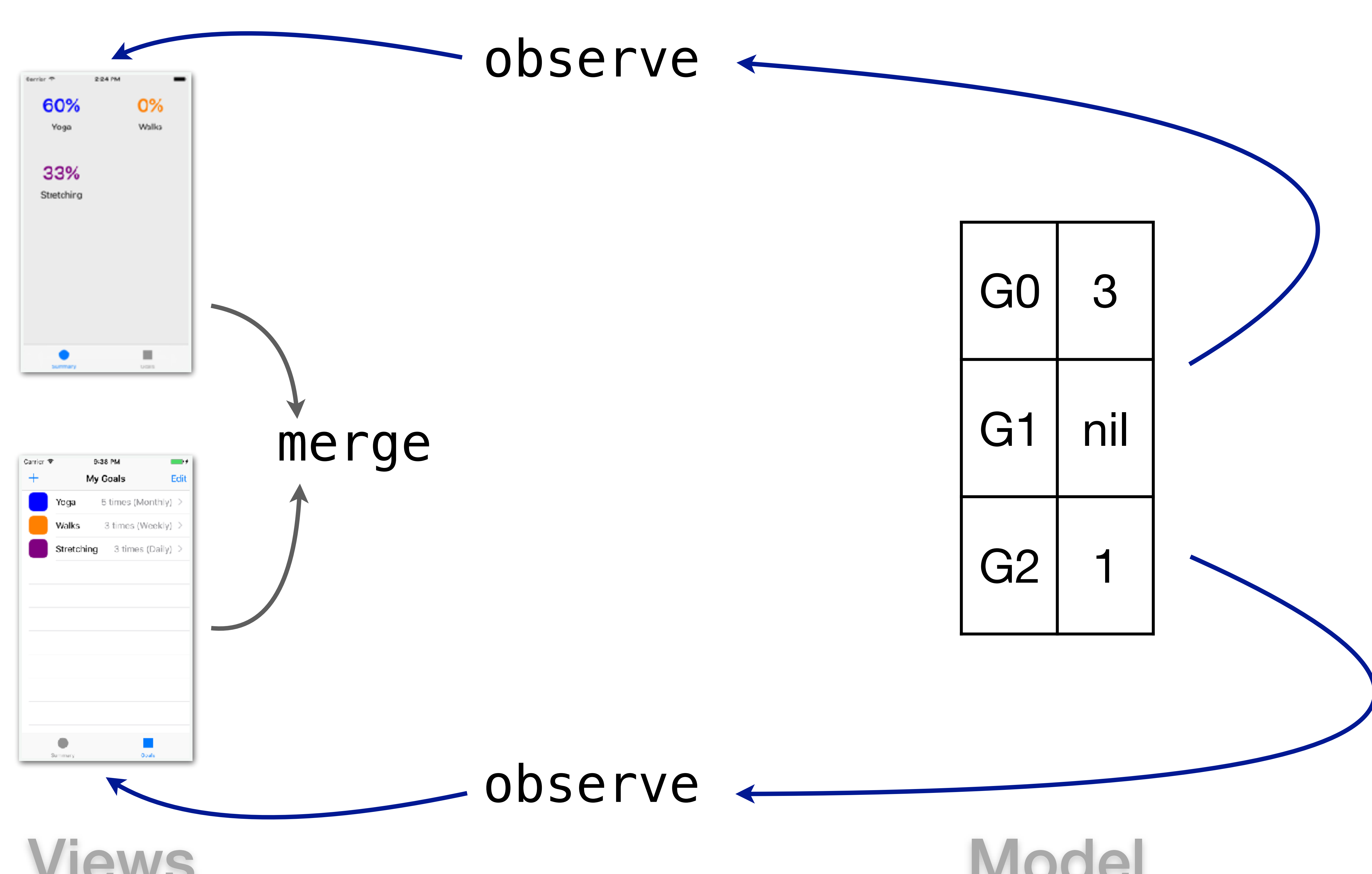
Views

Model

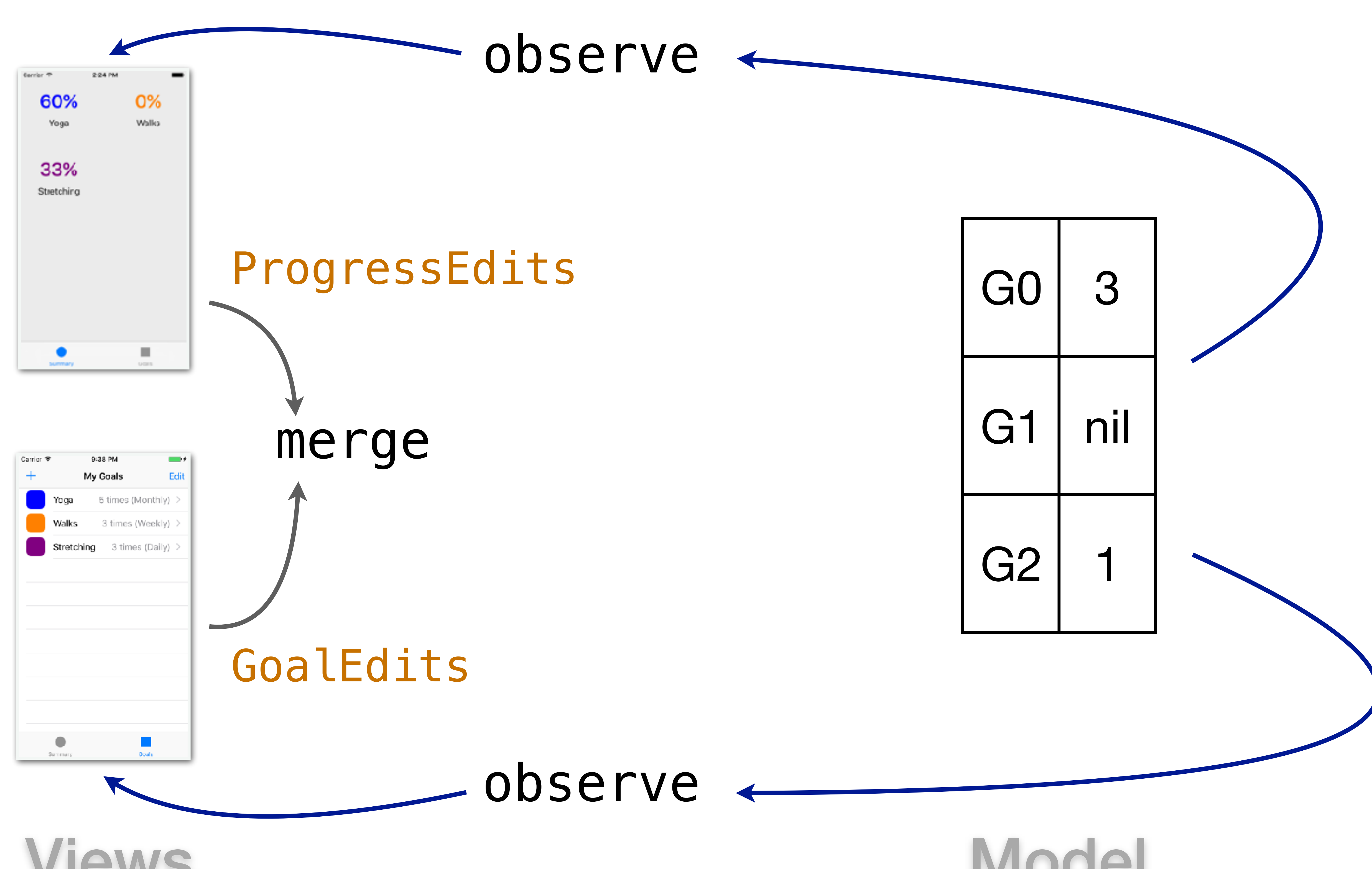
Change Propagation in Goals



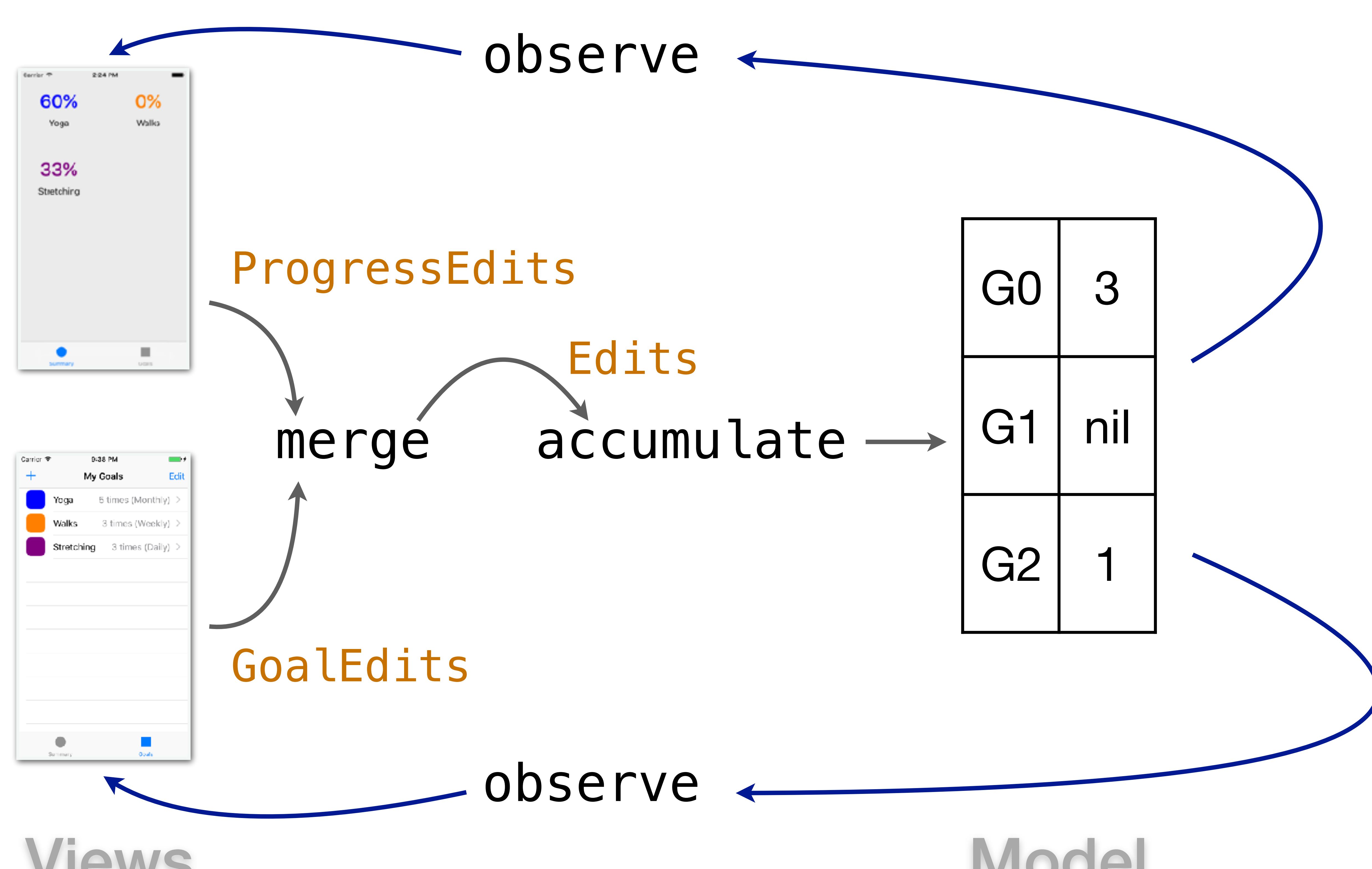
Change Propagation in Goals



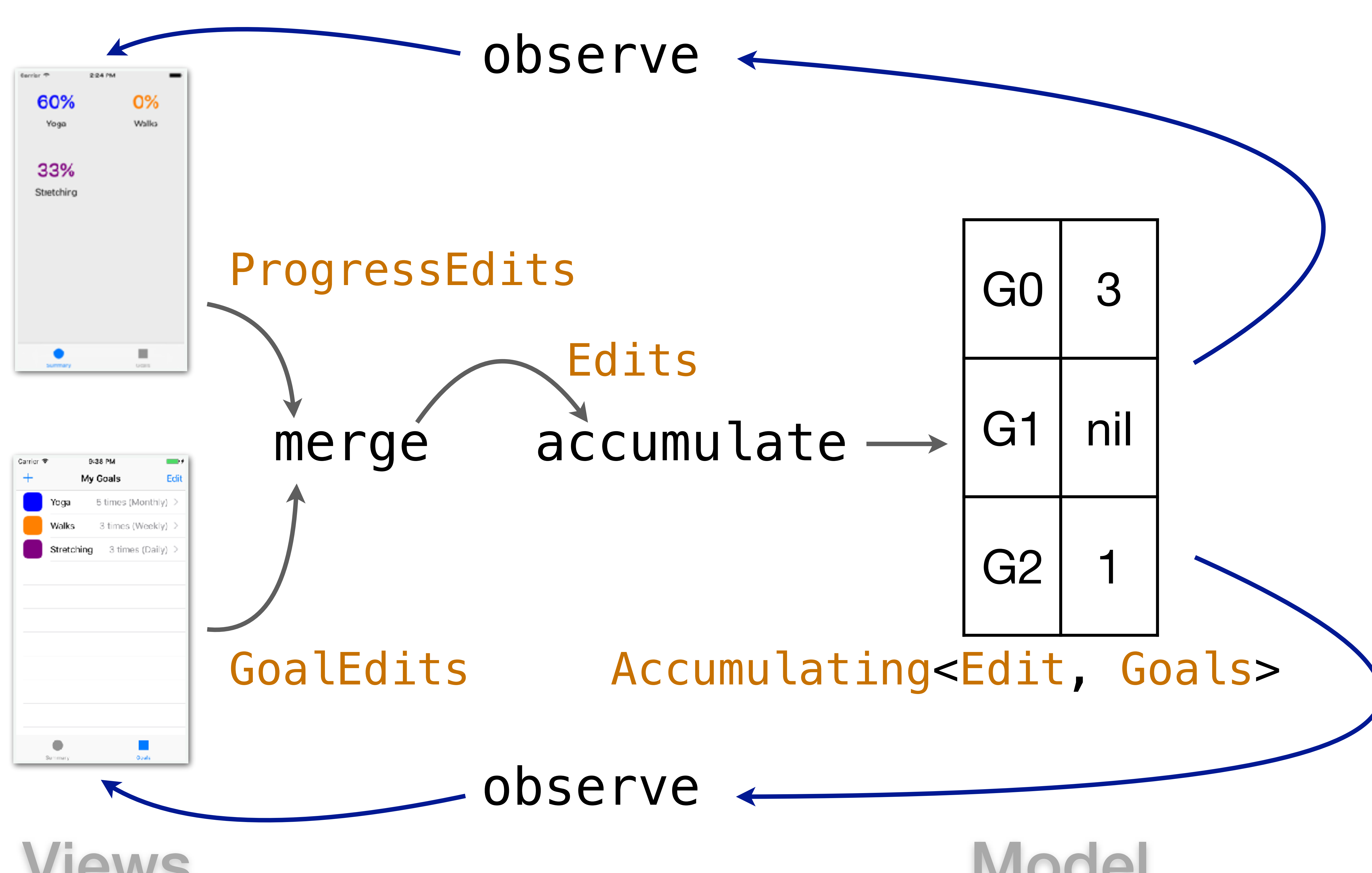
Change Propagation in Goals



Change Propagation in Goals



Change Propagation in Goals



```
GoalEdits

ProgressEdits

Edits

Accumulating<Edit, Goals>
```

```
ProgressEdits

Edits

Accumulating<Edit, Goals>
```

ProgressEdits

Edits

Accumulating<Edit, Goals>

```swift
typealias GoalEdits = Changing<GoalEdit>

enum GoalEdit {
  case add(goal: Goal)
  case delete(goal: Goal)
  case update(goal: Goal)
  case setActivity(activity: [Bool])
}
```

`ProgressEdits`

`Edits`

`Accumulating<Edit, Goals>`

```
typealias GoalEdits = Changing<GoalEdit>

enum GoalEdit {
  case add(goal: Goal)
  case delete(goal: Goal)
  case update(goal: Goal)
  case setActivity(activity: [Bool])
}

extension GoalEdit {
 func transform(_ goals: Goals) -> Goals {
   switch self {
    case .add(let newGoal):
    …
```

ProgressEdits

Edits

Accumulating<Edit, Goals>

Kolin Edition

```
typealias GoalEdits = Changing<GoalEdit>

sealed class GoalEdit {
  data class Add(val goal: Goal) : GoalEdit()
  data class Delete(val goal: Goal) : GoalEdit()
  data class Update(val goal: Goal) : GoalEdit()
  data class SetActivity(val activity: Array<Boolean>)
               : GoalEdit()
}

fun GoalEdit.transform(goals: Goals): Goals {
  when(this) {
    is GoalEdit.Add -> { …}
    …
}
```

```
typealias GoalEdits = Changing<GoalEdit>

enum GoalEdit {
  case add(goal: Goal)
  case delete(goal: Goal)
  case update(goal: Goal)
  case setActivity(activity: [Bool])
}

extension GoalEdit {
 func transform(_ goals: Goals) -> Goals {
   switch self {
    case .add(let newGoal):
    …
```

ProgressEdits

Edits

Accumulating<Edit, Goals>

GoalEdits

```
typealias ProgressEdits = Changing<ProgressEdit>
```

Edits

Accumulating<Edit, Goals>

GoalEdits

```
typealias ProgressEdits = Changing<ProgressEdit>
```

Edits

Accumulating<Edit, Goals>

GoalEdits

```
typealias ProgressEdits = Changing<ProgressEdit>

enum ProgressEdit {
  case bump(goal: Goal)
}
```

Edits

Accumulating<Edit, Goals>

GoalEdits

```
typealias ProgressEdits = Changing<ProgressEdit>

enum ProgressEdit {
  case bump(goal: Goal)
}

extension ProgressEdit {
 func transform(_ goals: Goals) -> Goals { … }
}
```

Edits

Accumulating<Edit, Goals>

GoalEdits

```
typealias ProgressEdits = Changing<ProgressEdit>

enum ProgressEdit {
  case bump(goal: Goal)
}

extension ProgressEdit {
 func transform(_ goals: Goals) -> Goals { … }
}
```

```
typealias Edits = Changing<Edit>
```

Accumulating<Edit, Goals>

GoalEdits

```
typealias ProgressEdits = Changing<ProgressEdit>

enum ProgressEdit {
  case bump(goal: Goal)
}

extension ProgressEdit {
 func transform(_ goals: Goals) -> Goals { … }
}
```

```
typealias Edits = Changing<Edit>

enum Edit {
  case goalEdit(edit: GoalEdit)
  case progressEdit(edit: ProgressEdit)
}
```

Accumulating<Edit, Goals>

GoalEdits

ProgressEdits

Edits

```
let model: Accumulating<Edit, Goals>
```

GoalEdits

ProgressEdits

Edits

```
let model: Accumulating<Edit, Goals>
```

GoalEdits

ProgressEdits

Edits

```
let model: Accumulating<Edit, Goals>
  = edits.accumulate(startingFrom: initialGoals) {
      edit, currentGoals in
        return edit.transform(currentGoals)
    }
```

Kolin Edition

```
val model: Accumulating<Edit, Goals>
    = edits.accumulate(initial = initialGoals)
      { edit, currentGoals ->
          return edit.transform(currentGoals)
      }
```

Replacing Tests

Replacing Tests

✓ Avoid changes from unexpected places

Replaces integration tests

Replacing Tests

✓ Avoid changes from unexpected places

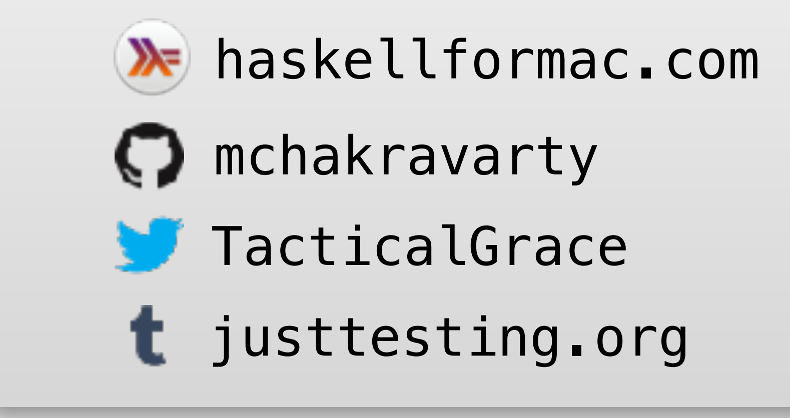
Replaces integration tests

✓ Avoid conflicting state changes

Replaces integration tests

Check out the source code of Goals.app!

https://github.com/mchakravarty/goalsapp



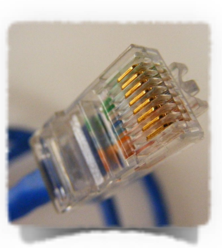
Check out the source code of Goals.app!

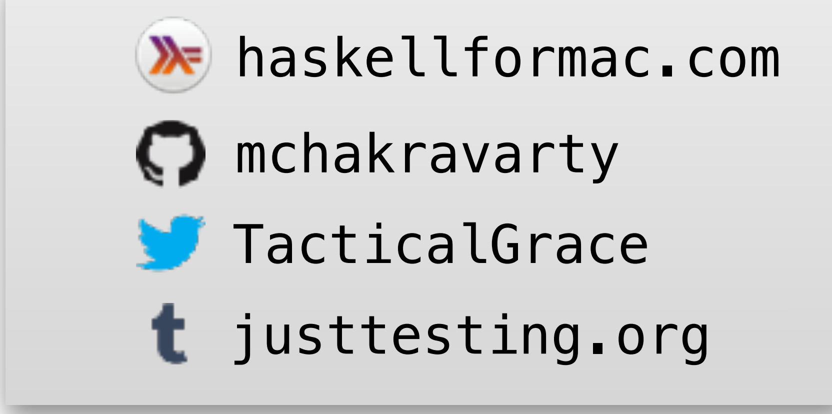
https://github.com/mchakravarty/goalsapp

Type systems are a design tool

Swift encourages typed functional programming

Types replace entire categories of tests





Thank you!

Image Attribution

https://www.flickr.com/photos/30998987@N03/5408763997
https://www.flickr.com/photos/provisions/7986149891/
https://www.flickr.com/photos/digitalcurrency/2438119267
https://www.flickr.com/photos/pviojoenchile/2760021279/

https://commons.wikimedia.org/wiki/File:Nile_blue_05.jpg
https://commons.wikimedia.org/wiki/File:Handcuffs01_2008-07-27.jpg
https://commons.wikimedia.org/wiki/File:Set_square_Geodreieck.svg

https://dribbble.com/shots/1667698-Free-vector-Macbook-Ipad-and-Iphone

https://openclipart.org/detail/463/heart