Functional Data Structures in Swift

Manuel M T Chakravarty
Applicative

@tacticalgrace.justtesting.org
@TacticalGrace
mchakravarty

cross-platform

open source

cross-platform

open source

cross-platform

multi-paradigm

open source

cross-platform

multi-paradigm

high-performance

strong functional core

open source

cross-platform

multi-paradigm

high-performance

”Name a defining characteristic of functional programming?”

”Name a defining characteristic of functional programming?”

Immutable data structures

”Name a defining characteristic of functional programming?”

Immutable data structures
aka value semantics

Hedging Against Mutability

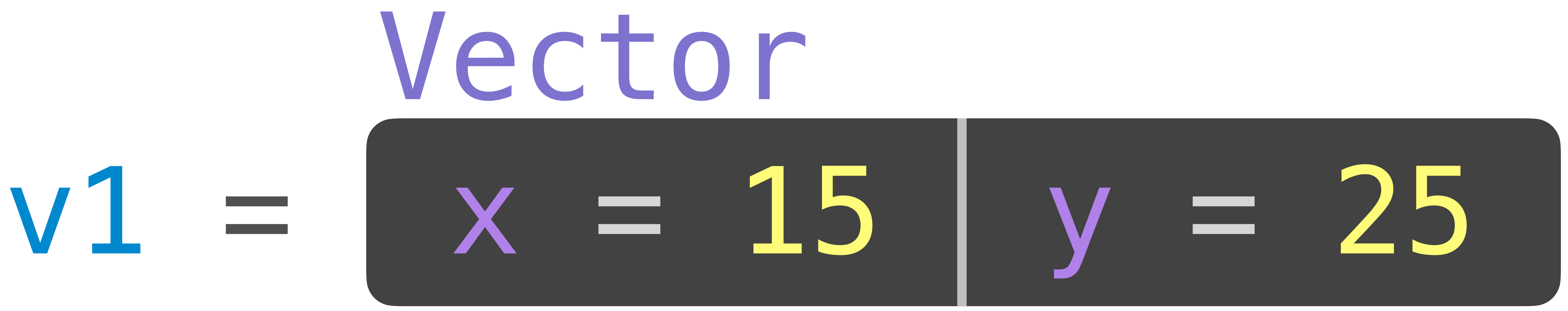
Value versus reference types

Value Semantics

Reference Semantics

Value Semantics

Reference Semantics



Value Semantics

Reference Semantics

```
v1 = Vector
     | x = 15 | y = 25 |

v2 = v1
```

Value Semantics

Reference Semantics

```
      Vector
v1 = | x = 15 | y = 25 |

      Vector
v2 = | x = 15 | y = 25 |
```

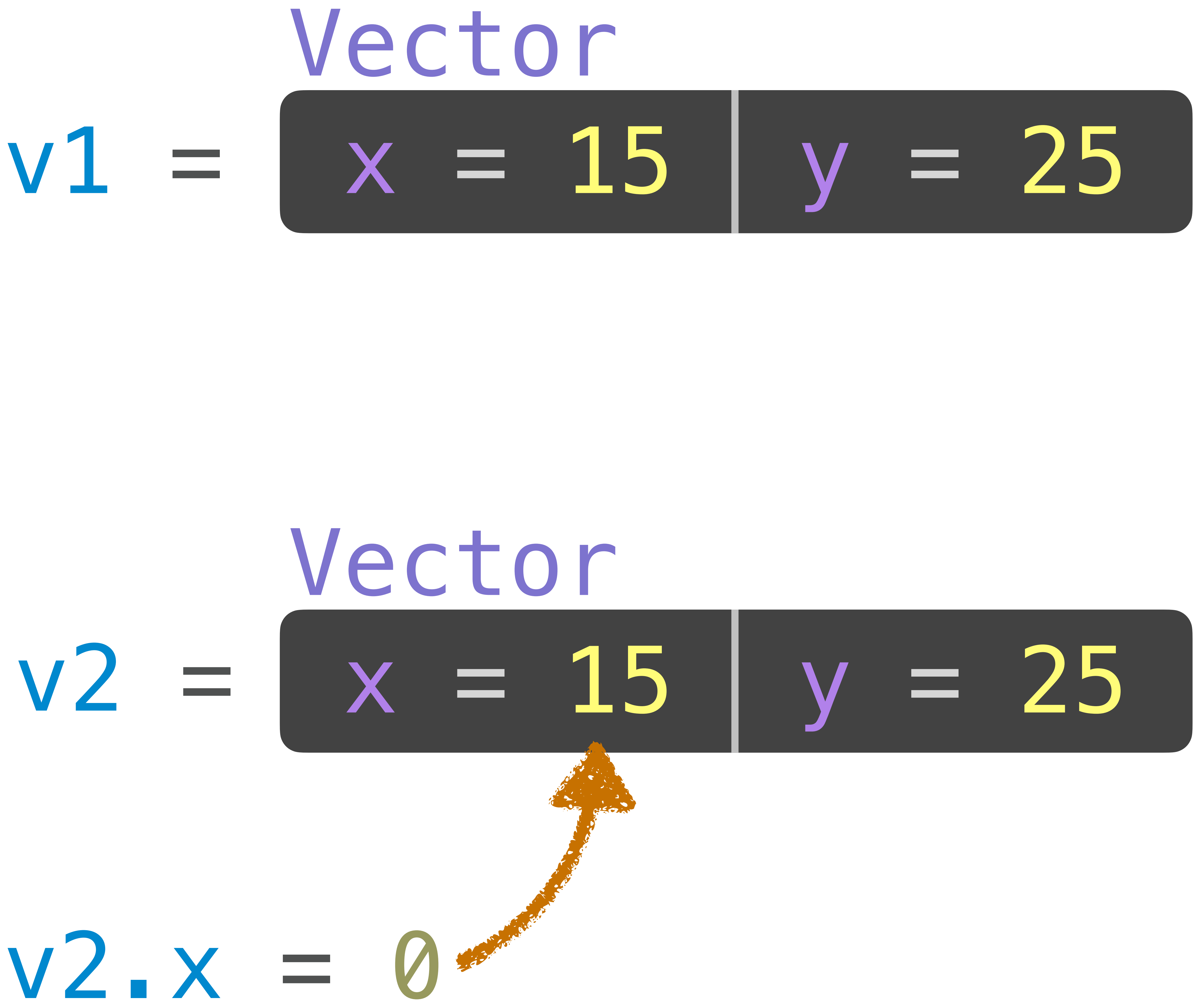
Value Semantics

Reference Semantics

```
      Vector
v1 = | x = 15 | y = 25 |

      Vector
v2 = | x = 15 | y = 25 |

v2.x = 0
```

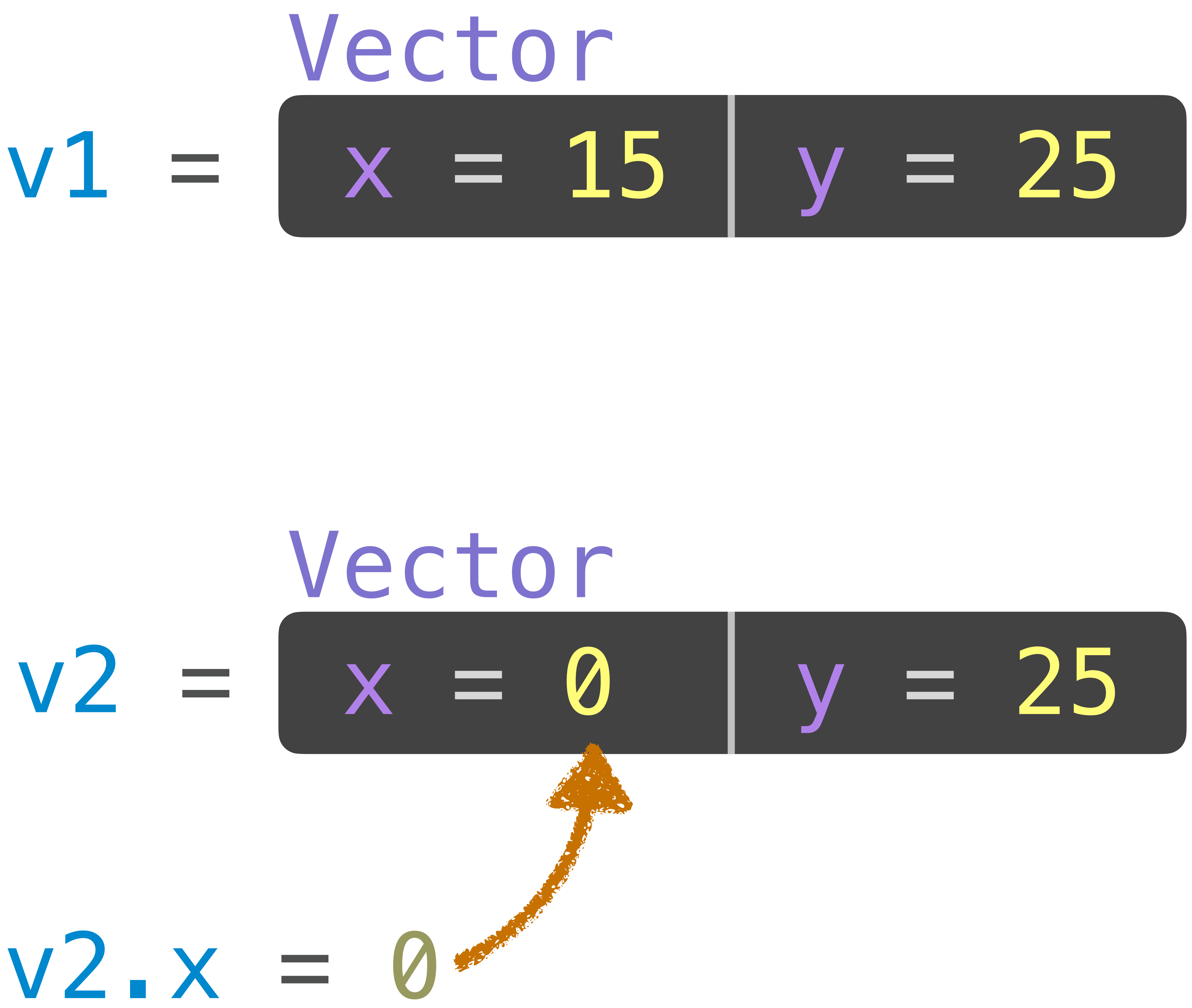
Value Semantics

Reference Semantics



Value Semantics

Reference Semantics



Value Semantics | Reference Semantics

```
     Vector
v1 = | x = 15 | y = 25 |

     Vector
v2 = | x = 0 | y = 25 |

v2.x = 0
```

Value Semantics

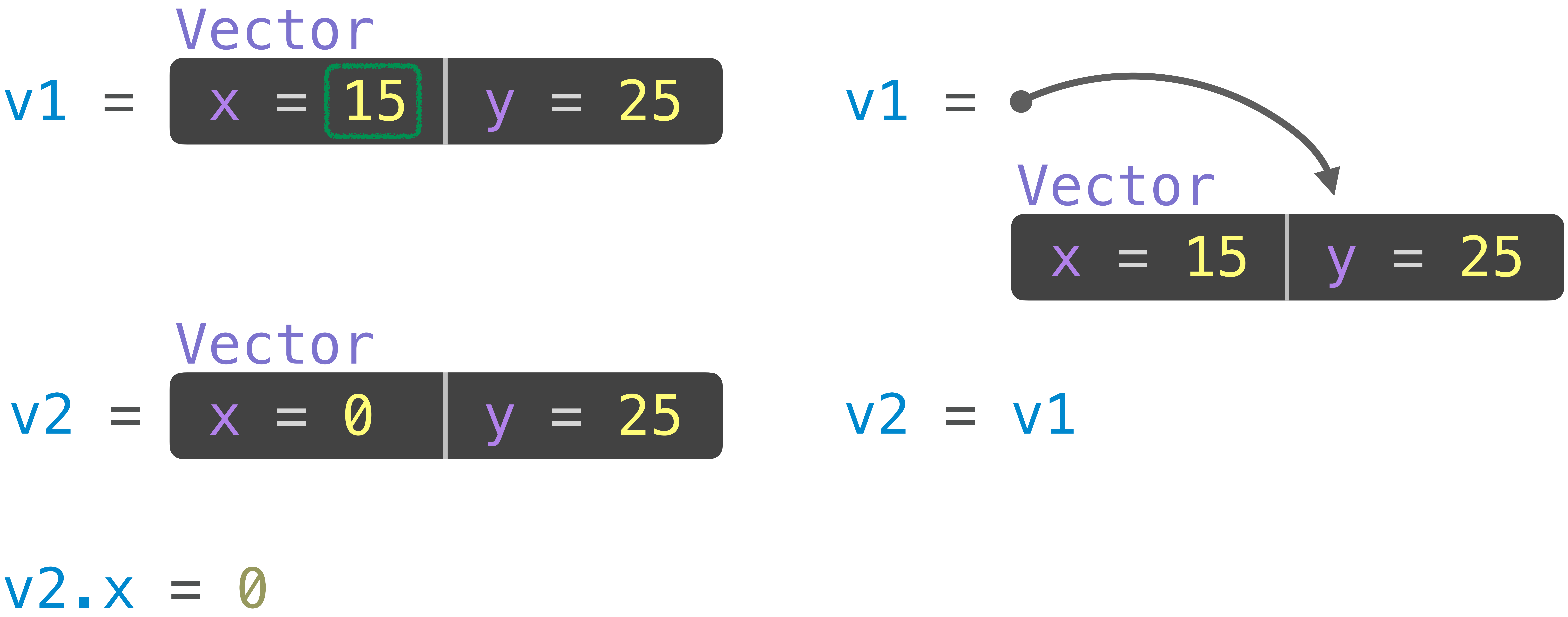
```
        Vector
v1 = [ x = 15 | y = 25 ]

        Vector
v2 = [ x = 0 | y = 25 ]

v2.x = 0
```

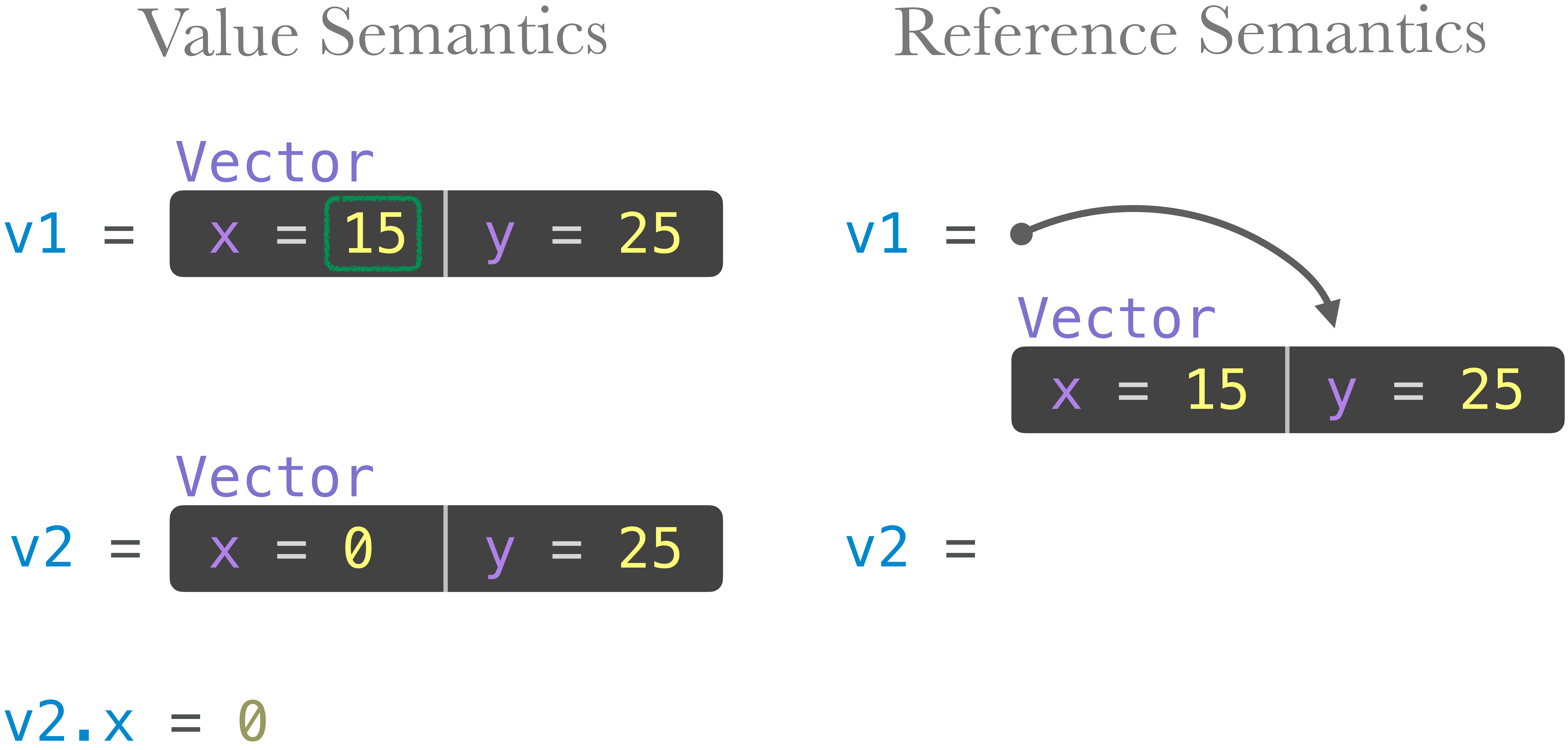
Reference Semantics

```
v1 = •───►
        Vector
       [ x = 15 | y = 25 ]
```

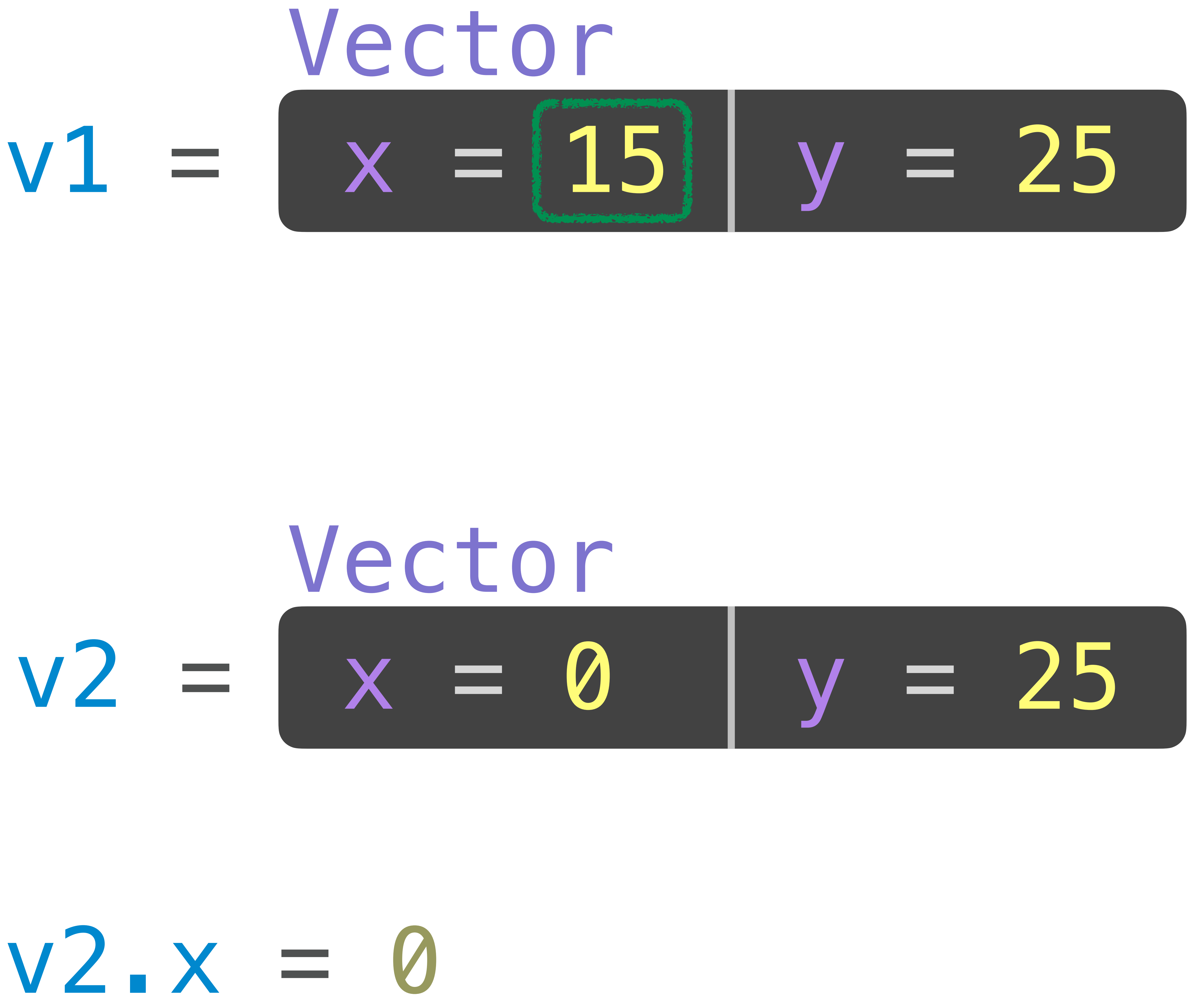
Value Semantics

Reference Semantics

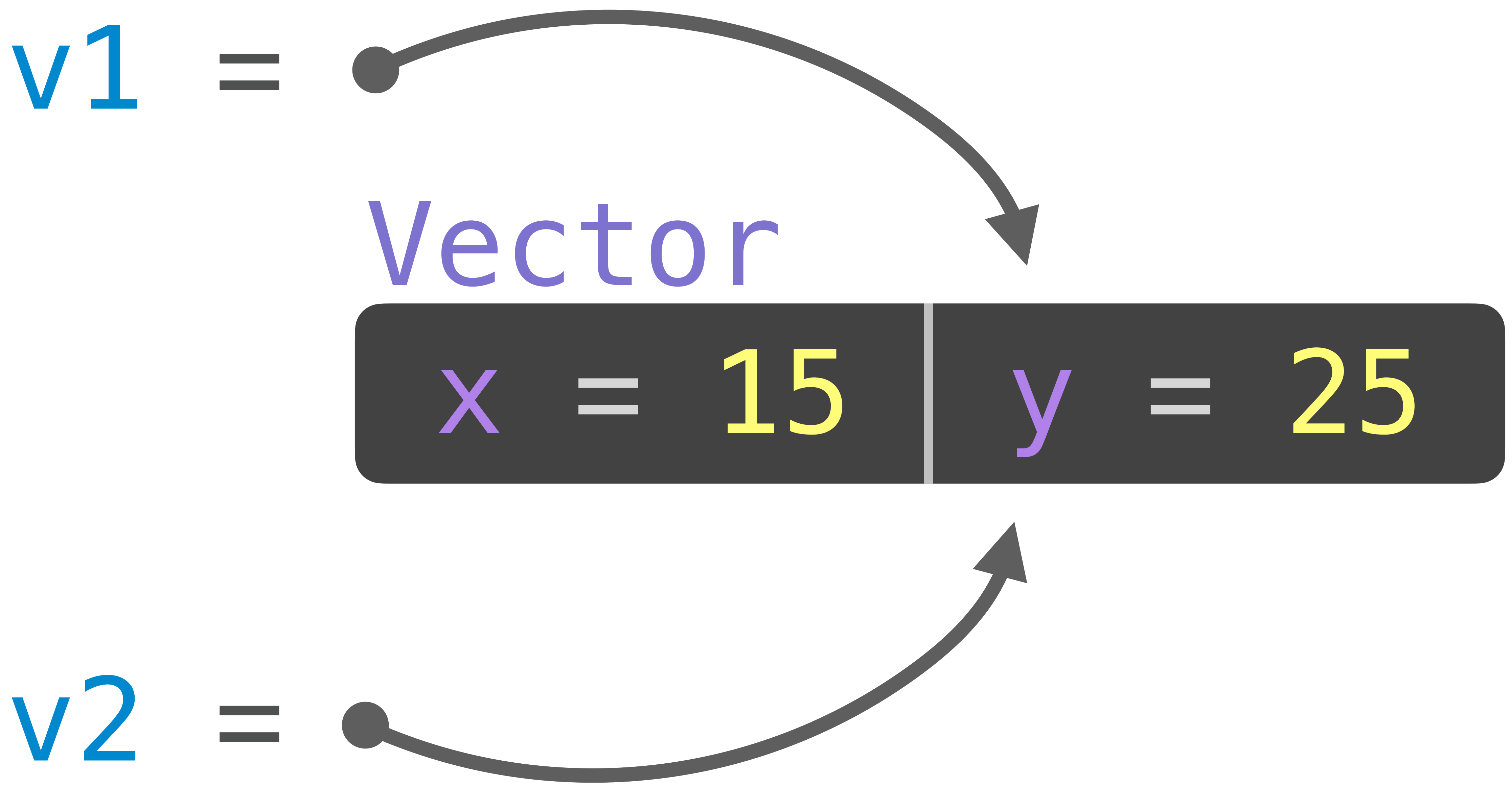



Value Semantics
Reference Semantics
Vector
v1 =
x = 15
y = 25
Vector
v2 =
x = 0
y = 25
v2.x = 0
v1 =
Vector
x = 15
y = 25
v2 =

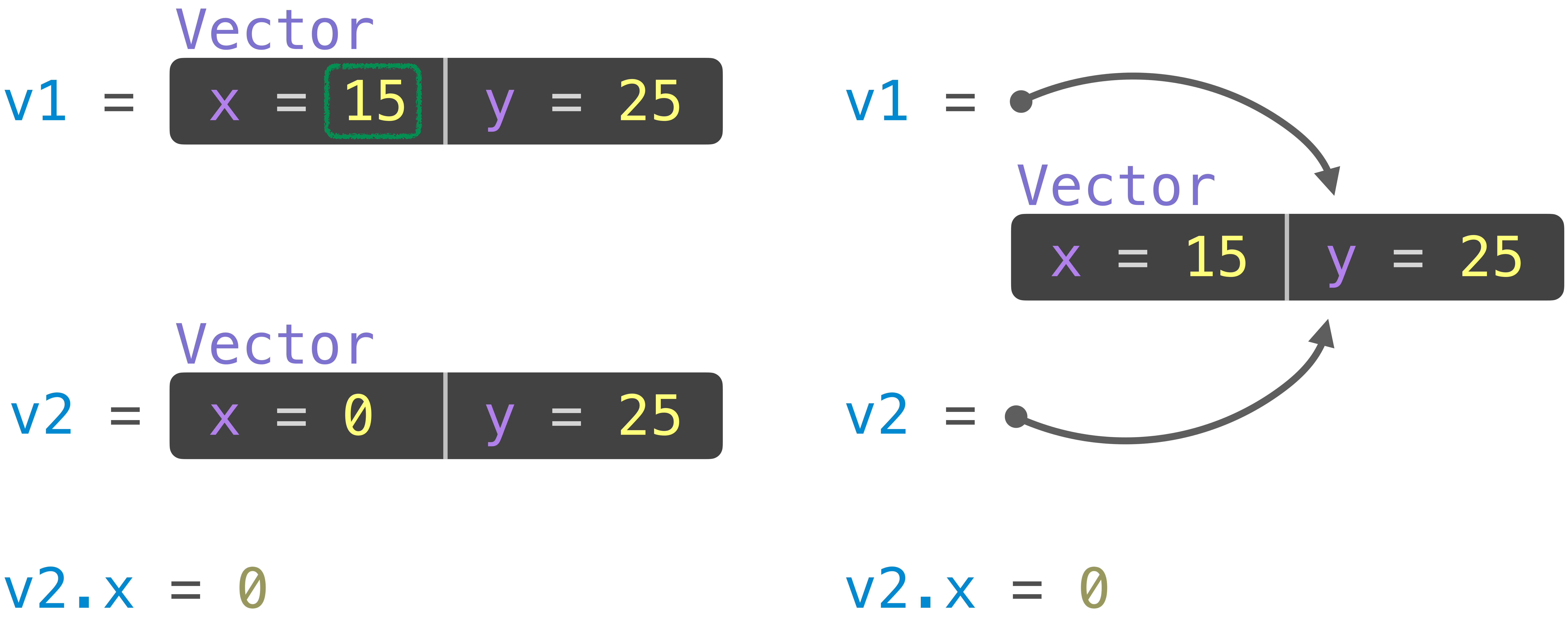
Value Semantics

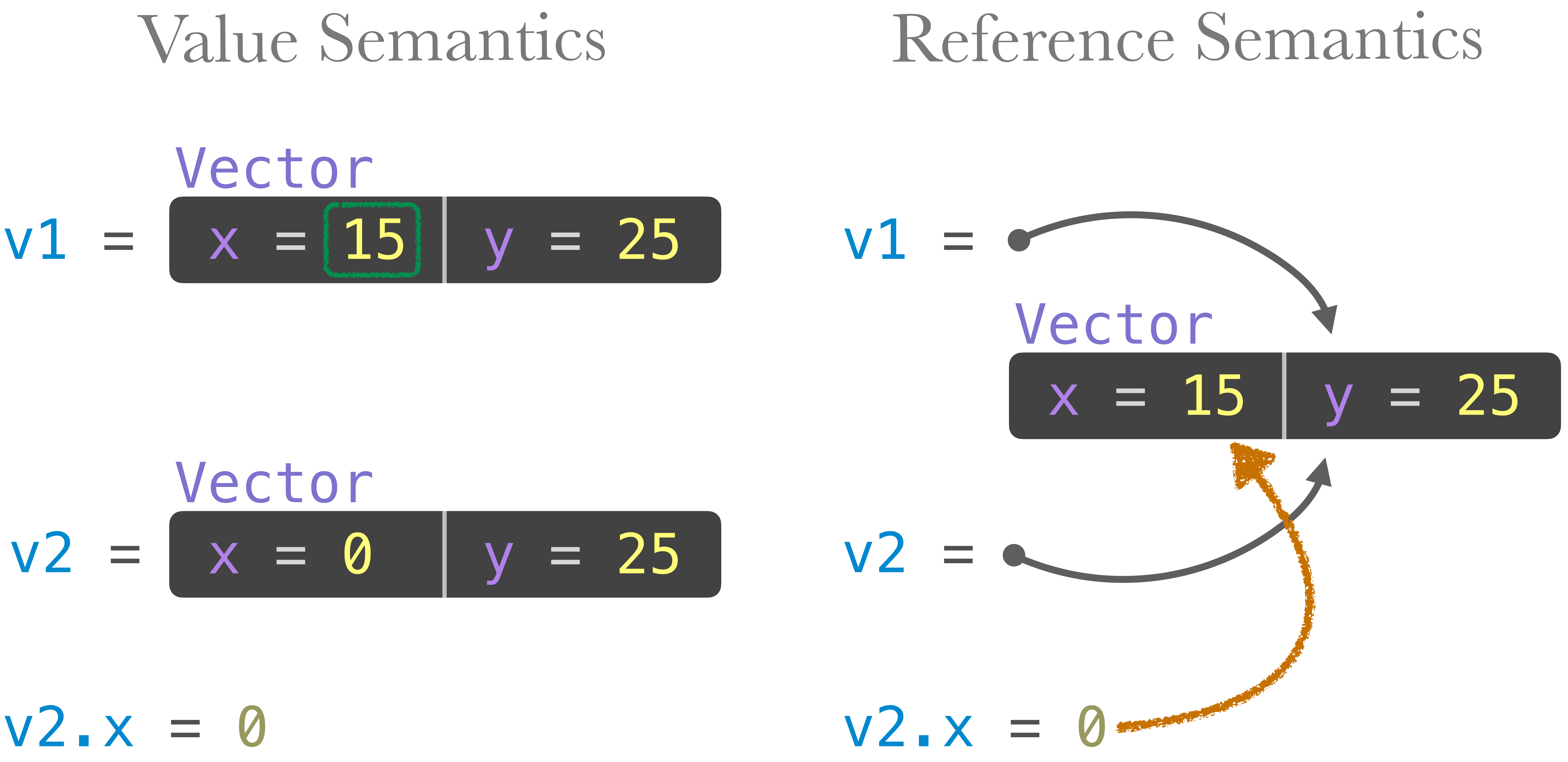


Reference Semantics

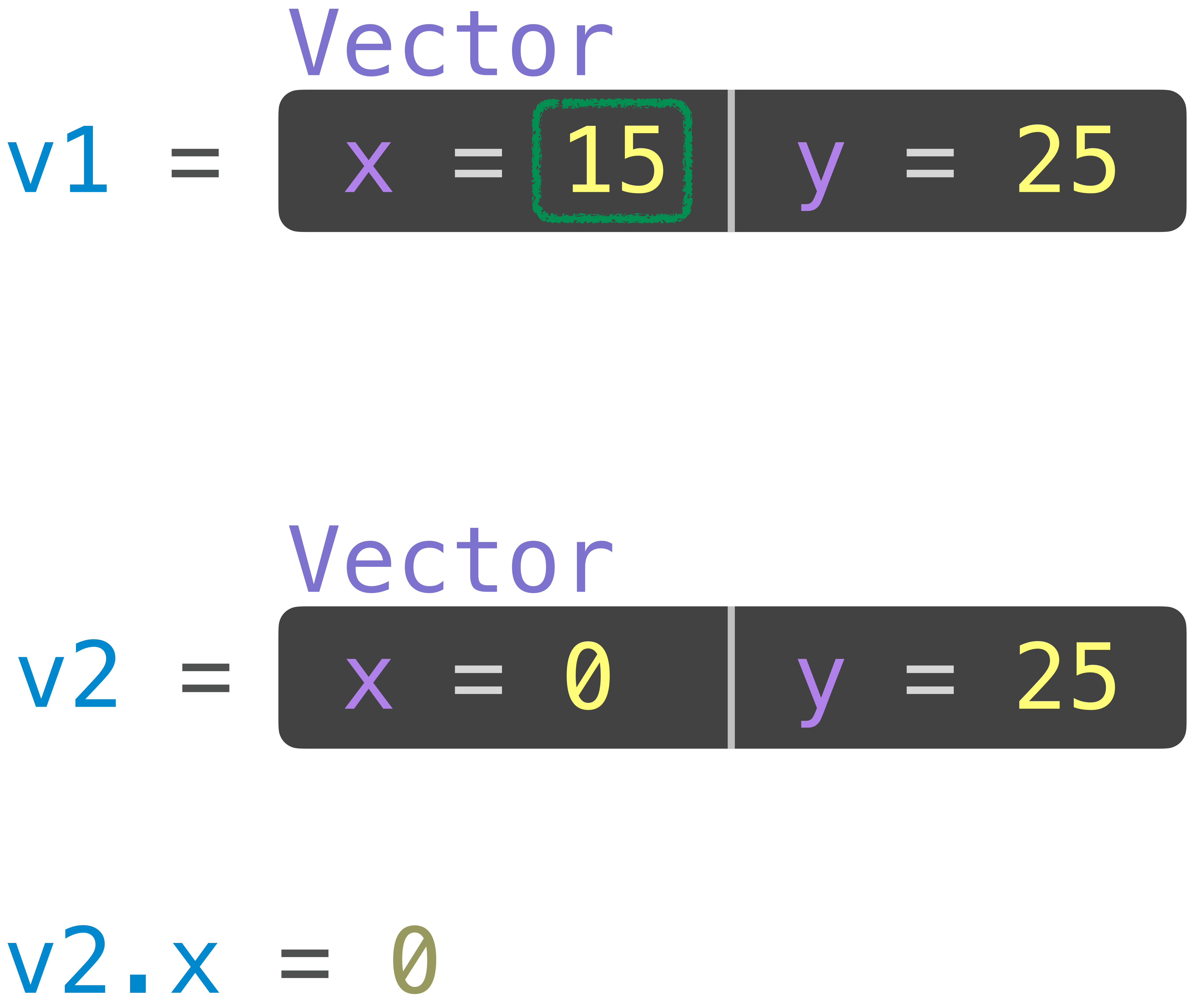


Value Semantics | Reference Semantics

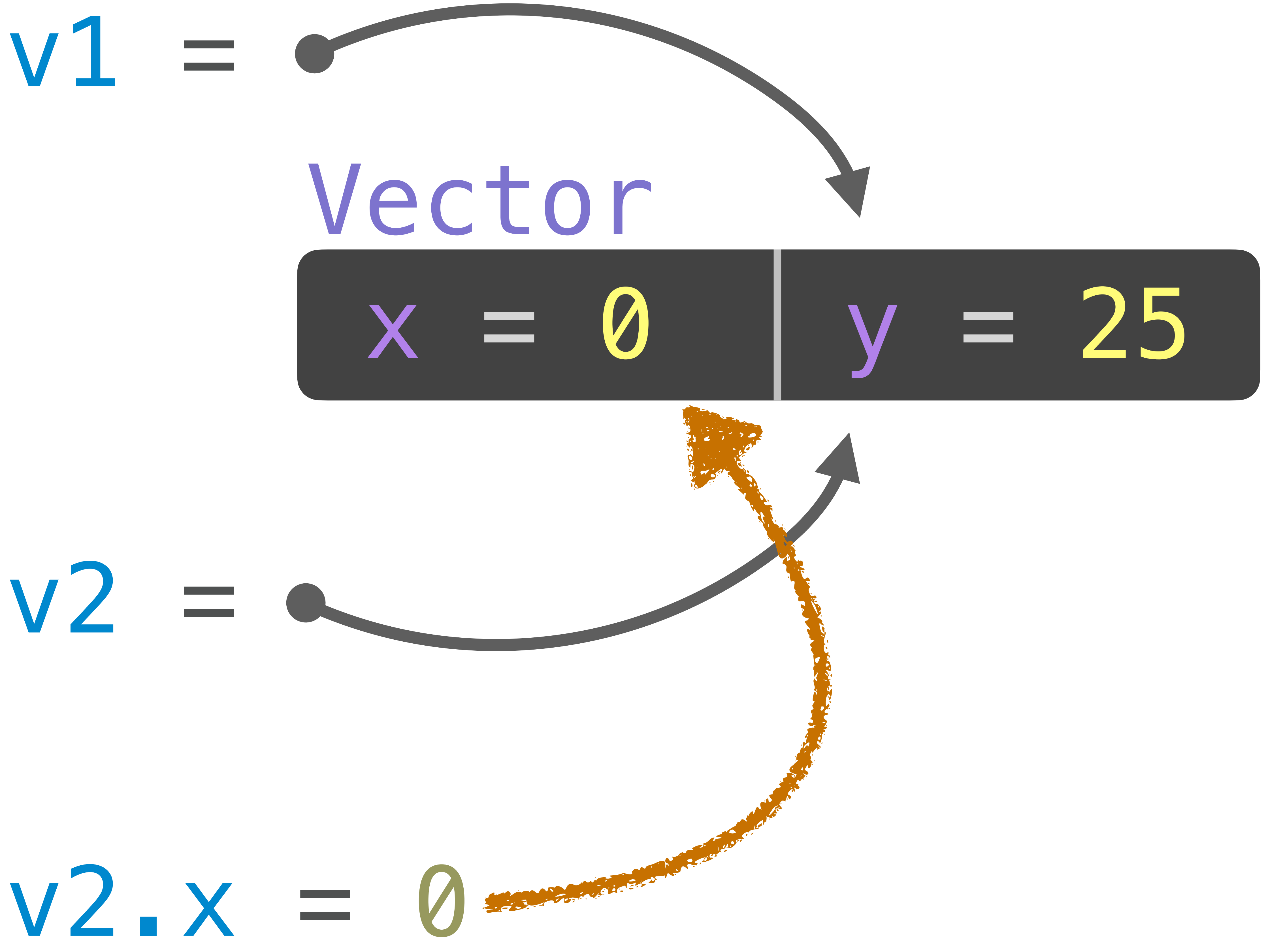



Value Semantics
Reference Semantics
Vector
v1 = x = 15 y = 25
Vector
v2 = x = 0 y = 25
v2.x = 0
v1 =
Vector
x = 15 y = 25
v2 =
v2.x = 0

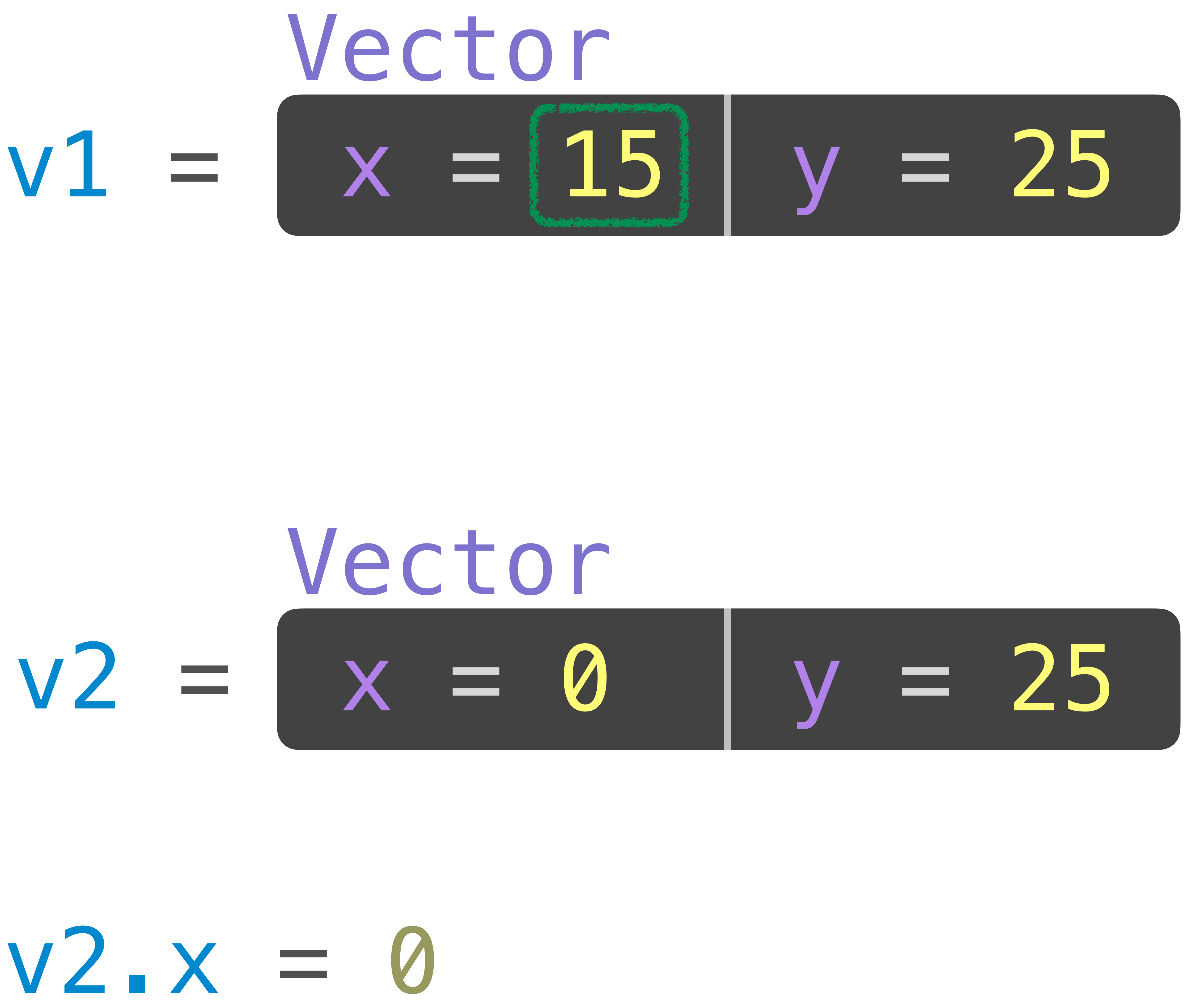
Value Semantics



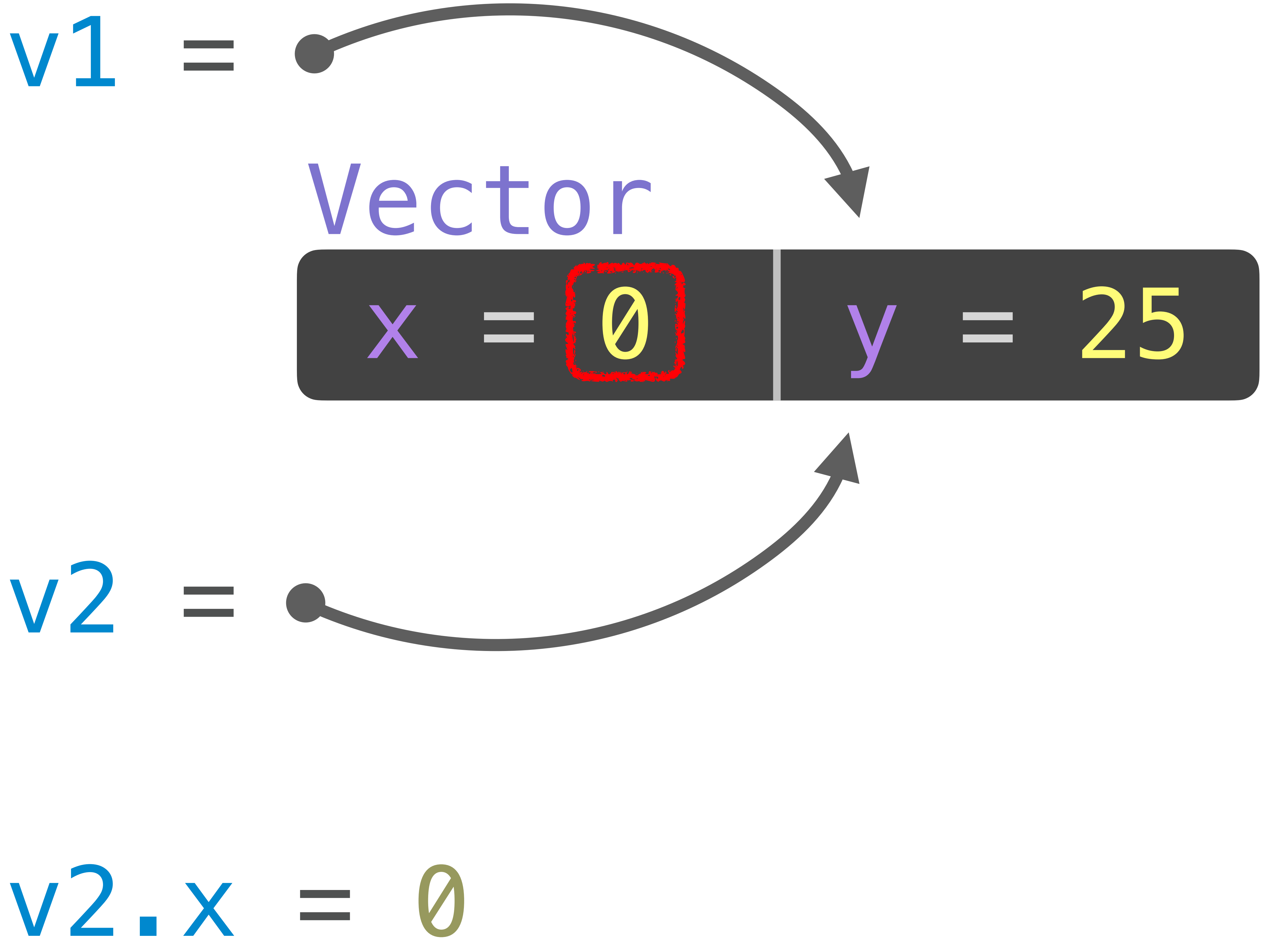
Reference Semantics

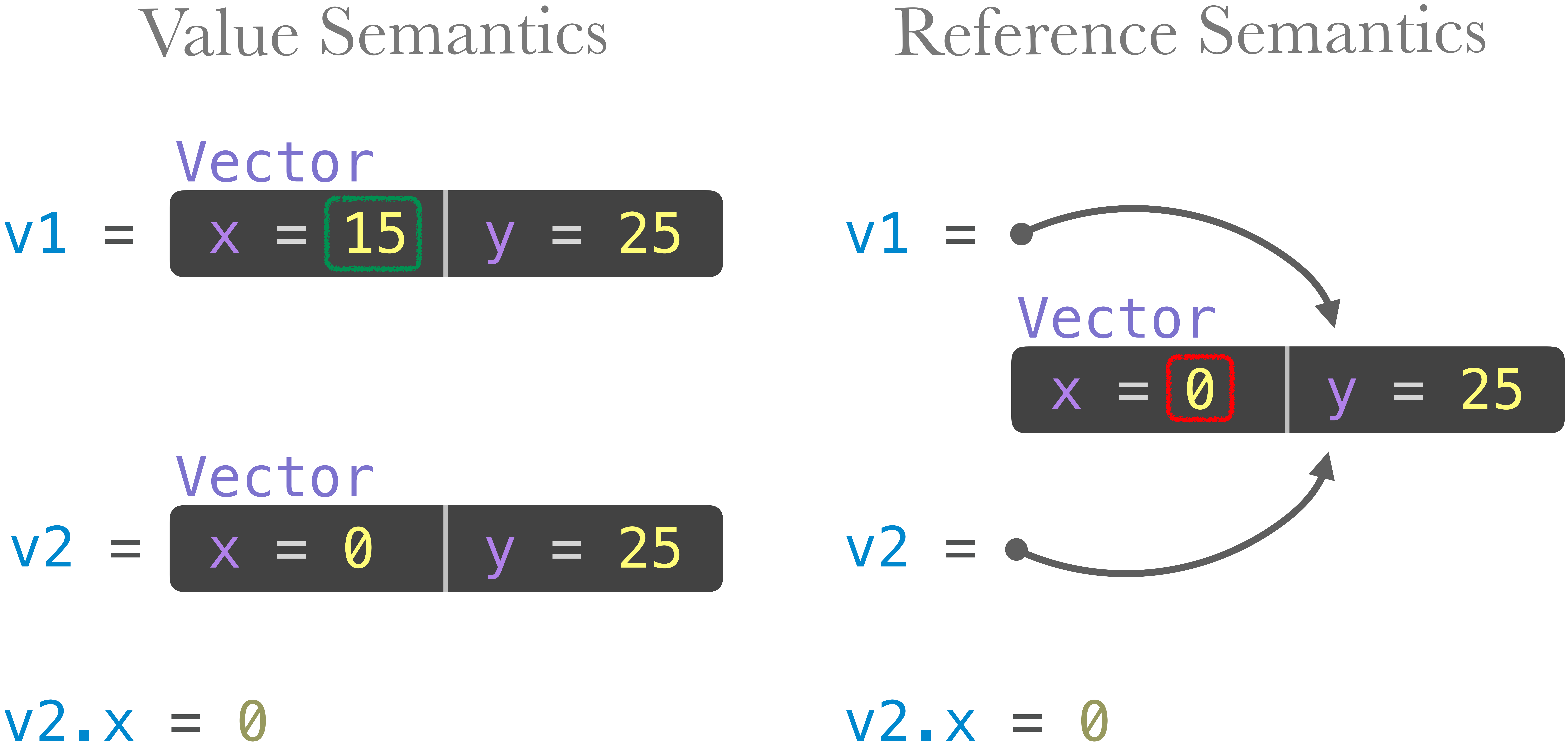


Value Semantics



Reference Semantics





Same when passing an argument to a function

Value types have value semantics.

Value types have value semantics.

Reference types have reference semantics.

Value types | Reference types

Value types

Reference types

```
struct MyStruct {
  var prop: SomeType
  …
}
```

Value types

Reference types

```
struct MyStruct {
  var prop: SomeType
  …
}
```

```
enum MyEnum {
  case firstVariant
  …
}
```

Value types

```swift
struct MyStruct {
  var prop: SomeType
  …
}
```

```swift
enum MyEnum {
  case firstVariant
  …
}
```

Reference types

```swift
class MyClass {
  var prop: SomeType
  …
}
```

Value types

```swift
struct MyStruct {
  var prop: SomeType
  …
}
```

```swift
enum MyEnum {
  case firstVariant
  …
}
```

Reference types

```swift
class MyClass {
  var prop: SomeType
  …
}
```

```swift
actor MyActor {
  var prop: SomeType
  …
}
```

Value types

```
struct MyStruct {
  var prop: SomeType
  …
}
```

```
enum MyEnum {
  case firstVariant
  …
}
```

Reference types

```
class MyClass {
  var prop: SomeType
  …
}
```

```
actor MyActor {
  var prop: SomeType
  …
}
```

The type declaration fixes the semantics.

Value and Reference Semantics in Code

Value type

```
struct Vector {
  var x: Float
  var y: Float
}
```

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
```

Value and Reference Semantics in Code

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
```

```
class VectorC {
  var x: Float
  var y: Float
}
```

Value and Reference Semantics in Code

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
```

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
```

Value and Reference Semantics in Code

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1
```

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
```

Value and Reference Semantics in Code

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
```

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
```

Value and Reference Semantics in Code

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
```

Value and Reference Semantics in Code

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}

var v1 = VectorC(x: 15,
                 y: 25)
```

Value and Reference Semantics in Code

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}

var v1 = VectorC(x: 15,
                 y: 25)
var v2 = v1
```

Value and Reference Semantics in Code

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}

var v1 = VectorC(x: 15,
                 y: 25)
var v2 = v1

v2.x = 0
```

Value and Reference Semantics in Code

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}

var v1 = VectorC(x: 15,
                 y: 25)
var v2 = v1

v2.x = 0
print(v1)
] VectorC(x: 0, y: 25)
```

reference gets copied

Value and Reference Semantics in Code

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

let v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}

let v1 = VectorC(x: 15,
                 y: 25)
let v2 = v1

v2.x = 0
print(v1)
] VectorC(x: 0, y: 25)
```

reference gets copied

Value and Reference Semantics in Code

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

let v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}

let v1 = VectorC(x: 15,
                 y: 25)
let v2 = v1

v2.x = 0
print(v1)
] VectorC(x: 0, y: 25)
```

reference gets copied

Value and Reference Semantics in Code

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

let v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}

let v1 = VectorC(x: 15,
                 y: 25)
let v2 = v1

v2.x = 0
print(v1)
] VectorC(x: 0, y: 25)
```

reference gets copied

Scaling up

Bulk value types

Structure

String

A Unicode string value that is a collection of characters.

Declaration

```
@frozen struct String
```

Structure

String

A Unicode string value that is a collection of characters.

Declaration

```
@frozen struct String
```

```
let hello: String = "Hello"
```

Structure

String

A Unicode string value that is a collection of characters.

Declaration

```
@frozen struct String
```

```
let hello: String = "Hello"
hello += ” World!"
```

Left side of mutating operator isn't mutable: 'hello' is a 'let' constant

Structure

String

A Unicode string value that is a collection of characters.

Declaration

```
@frozen struct String
```

```
let hello: String = "Hello"
```

Left side of mutating operator isn't mutable: 'hello' is a 'let' constant

Structure

String

A Unicode string value that is a collection of characters.

Declaration

```
@frozen struct String
```

```swift
let hello: String = "Hello"
var helloWorld = hello
```

Structure

String

A Unicode string value that is a collection of characters.

Declaration

```
@frozen struct String
```

```
let hello: String = "Hello"
var helloWorld = hello
helloWorld += ” World!"
```

Structure

String

A Unicode string value that is a collection of characters.

Declaration

```
@frozen struct String
```

```
let hello: String = "Hello"
var helloWorld = hello
helloWorld += ” World!"
print(hello)
] ”Hello"
```

Clearly, value semantics!

Interlude: Strings in Haskell

Interlude: Strings in Haskell

```
String
```

Interlude: Strings in Haskell

`String`

`ByteString`

Interlude: Strings in Haskell

`String`

`ByteString`

`Text`

Tradeoffs between convenience, performance, and efficiency.

Interlude: Strings in Haskell

`String`

`Text`

`ByteString`

Tradeoffs between convenience, performance, and efficiency.

And that's not even sufficient…

Interlude: Strings in Haskell

`String`

`Text`

`ByteString`

Tradeoffs between convenience, performance, and efficiency.

And that's not even sufficient…

`MVector`

(in a state monad?)

Structure

Array

An ordered, random-access collection.

Declaration

```
@frozen struct Array<Element>
```

```
let arr = [1, 2, 3, 4, 5, 6]
```

Structure

Array

An ordered, random-access collection.

Declaration

```
@frozen struct Array<Element>
```

```
let arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }          // ⇒ [3, 4, 5, 6, 7, 8]
```

Structure

Array

An ordered, random-access collection.

Declaration

```
@frozen struct Array<Element>
```

```
let arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }            // ⇒ [3, 4, 5, 6, 7, 8]
arr[2]                       // ⇒ 3
```

Structure

Array

An ordered, random-access collection.

Declaration

```
@frozen struct Array<Element>
```

```
let arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }          // ⇒ [3, 4, 5, 6, 7, 8]
arr[2]                     // ⇒ 3
arr[2] = 10
```

Cannot assign through subscript: 'arr' is a 'let' constant

Structure

Array

An ordered, random-access collection.

Declaration

```
@frozen struct Array<Element>
```

```
var arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }            // ⇒ [3, 4, 5, 6, 7, 8]
arr[2]                       // ⇒ 3
arr[2] = 10
```

Cannot assign through subscript: 'arr' is a 'let' constant

Structure

Array

An ordered, random-access collection.

Declaration

```
@frozen struct Array<Element>
```

```
var arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }          // ⇒ [3, 4, 5, 6, 7, 8]
arr[2]                     // ⇒ 3
arr[2] = 10
```

```swift
func printShuffled(_ arr: [Int]) {
   var localArr = arr
   localArr.shuffle()
   print(localArr)
}
```

```swift
var arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }          // ⇒ [3, 4, 5, 6, 7, 8]
arr[2]                     // ⇒ 3
arr[2] = 10                // == [1, 2, 10, 4, 5, 6]
printShuffled(arr)         // unchanged!
```

```swift
func printShuffled(_ arr: [Int]) {
    var localArr = arr
    localArr.shuffle()
    print(localArr)
 }
```

changes local copy only

```swift
var arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }          // ⇒ [3, 4, 5, 6, 7, 8]
arr[2]                     // ⇒ 3
arr[2] = 10                // == [1, 2, 10, 4, 5, 6]
printShuffled(arr)         // unchanged!
```

Mutable array, but passed by value

Structure

Dictionary

A collection whose elements are key-value pairs.

Declaration

```
@frozen struct Dictionary<Key, Value> where Key : Hashable
```

Structure

Dictionary

A collection whose elements are key-value pairs.

Declaration

```
@frozen struct Dictionary<Key, Value> where Key : Hashable
```

A dictionary is essential a hash table

Structure

Dictionary

A collection whose elements are key-value pairs.

Declaration

```
@frozen struct Dictionary<Key, Value> where Key : Hashable
```

A dictionary is essential a hash table

with value semantics

Structure

Dictionary

A collection whose elements are key-value pairs.

Declaration

```
@frozen struct Dictionary<Key, Value> where Key : Hashable
```

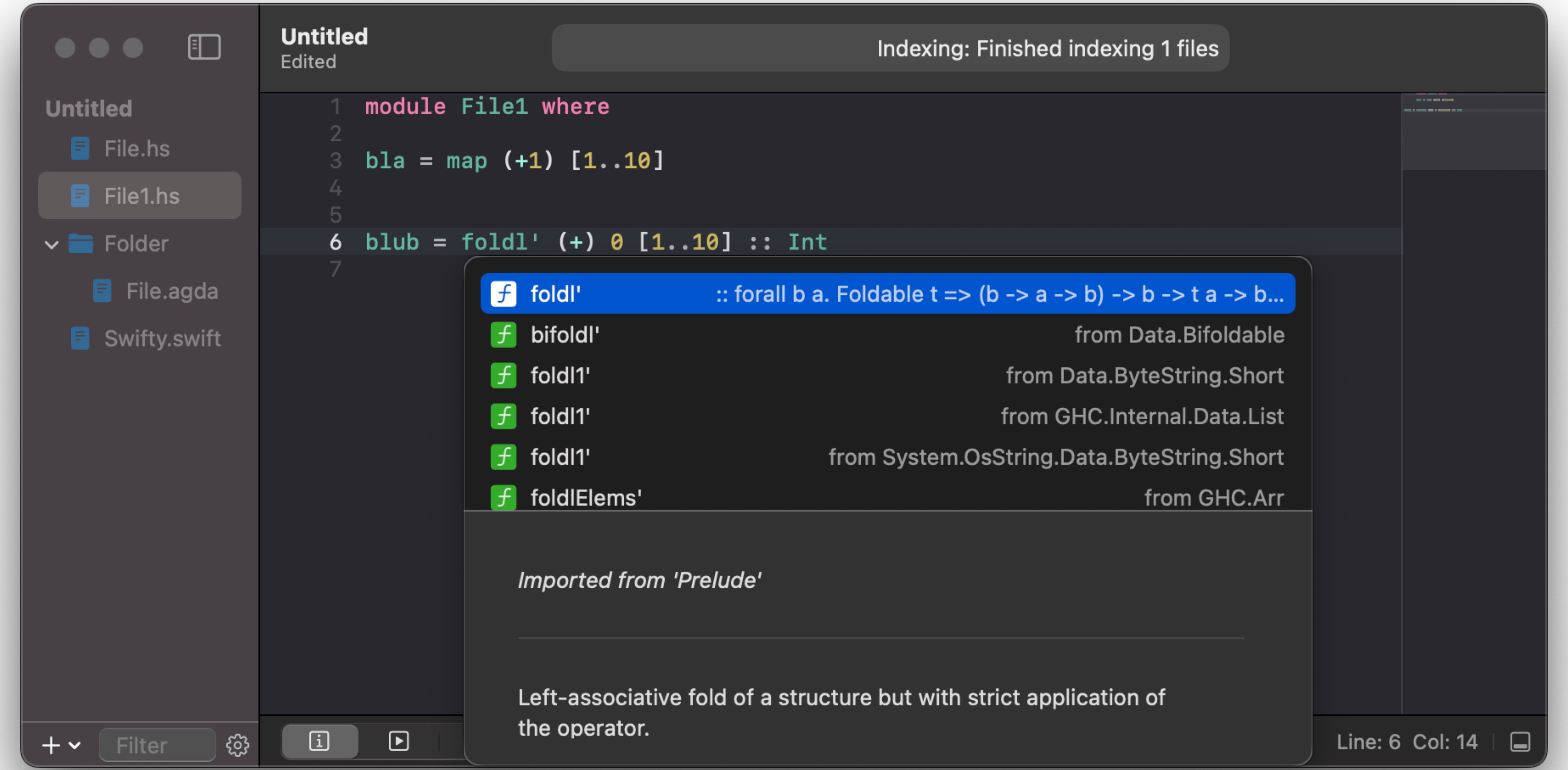
A dictionary is essential a hash table

with value semantics

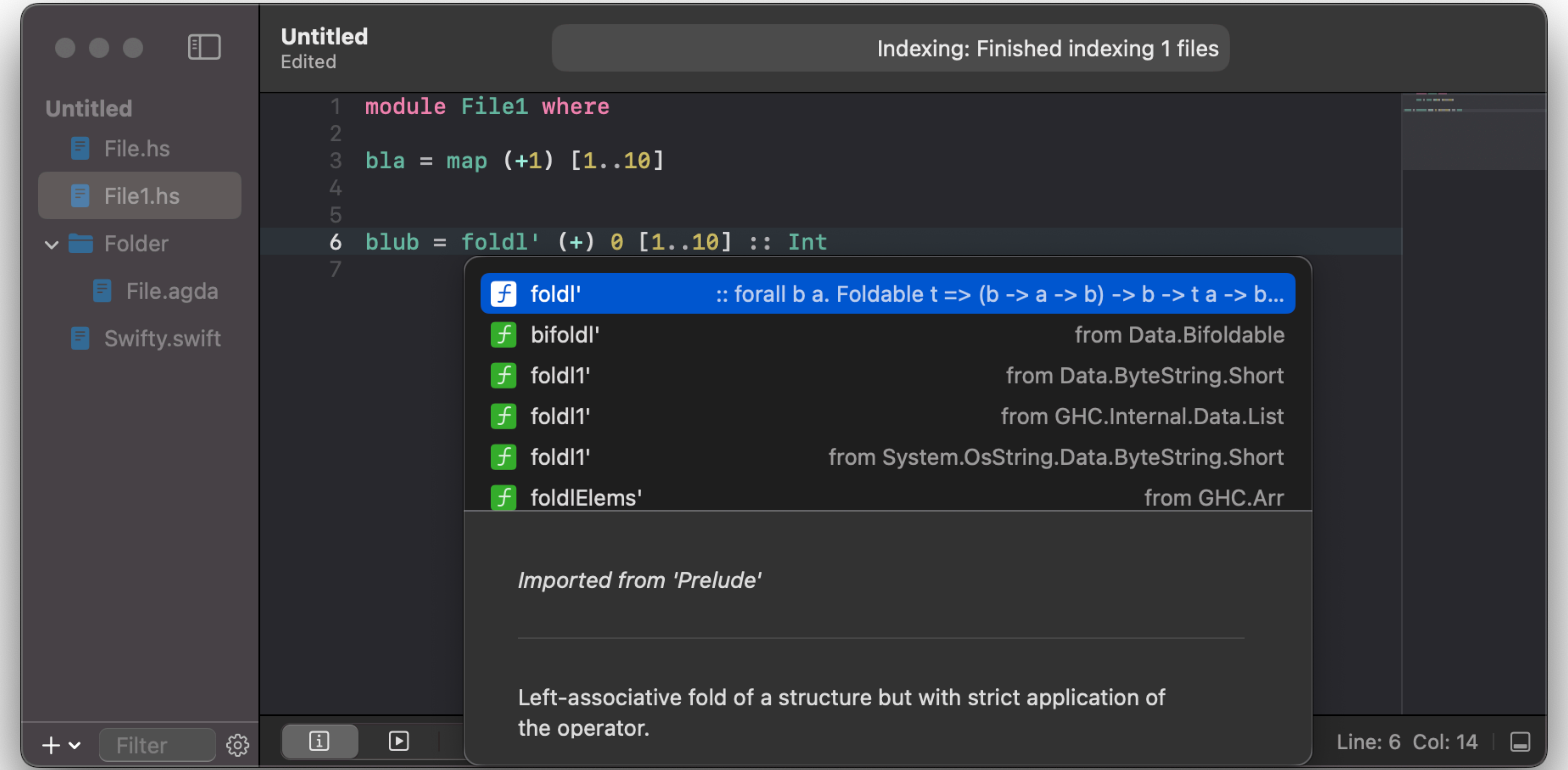
that can be immutable or mutable.

”But what about performance?”

Applicative Code



Applicative Code



Using strings for file contents and editor buffers

Requirements for Bulk Value Types

Requirements for Bulk Value Types

Preserve value semantics

Requirements for Bulk Value Types

Preserve value semantics

Minimise the number of copies made


Cow to the Rescue
Copy-on-write data structures

Payload

v1 = …data…

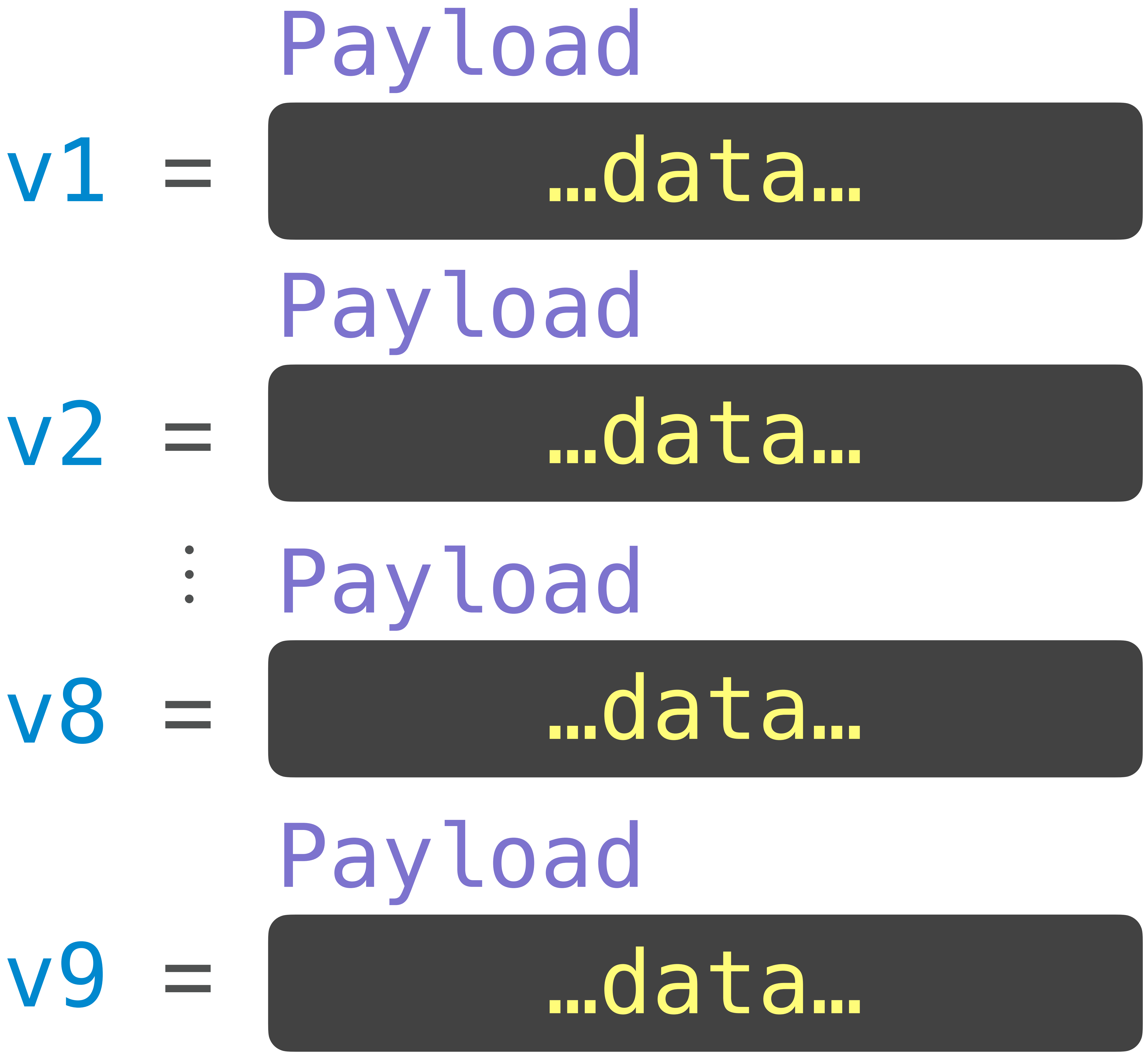
```
     Payload
v1 = …data…

v2 = v1
```

```
     Payload
v1 = …data…
     Payload
v2 = …data…
```

```
       Payload
v1 = [ …data… ]
       Payload
v2 = [ …data… ]
   ⋮
v8 = v7

v9 = v8
```

```
      Payload
v1 = [ …data… ]
      Payload
v2 = [ …data… ]
  ⋮   Payload
v8 = [ …data… ]
      Payload
v9 = [ …data… ]
```

That’s a lot of waste!



Payload
v1 =
…data…
Payload
v2 =
…data…
⋮
Payload
v8 =
…data…
Payload
v9 =
…data…

v1 =

v2 =

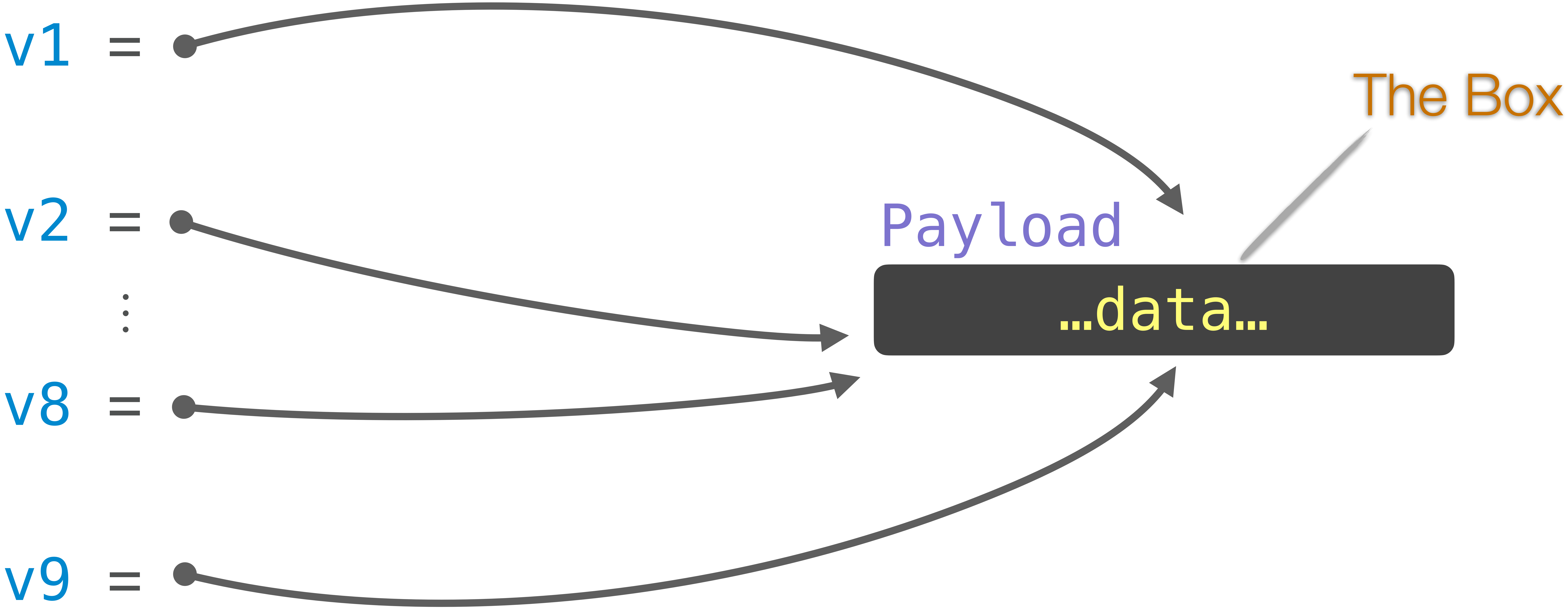
⋮

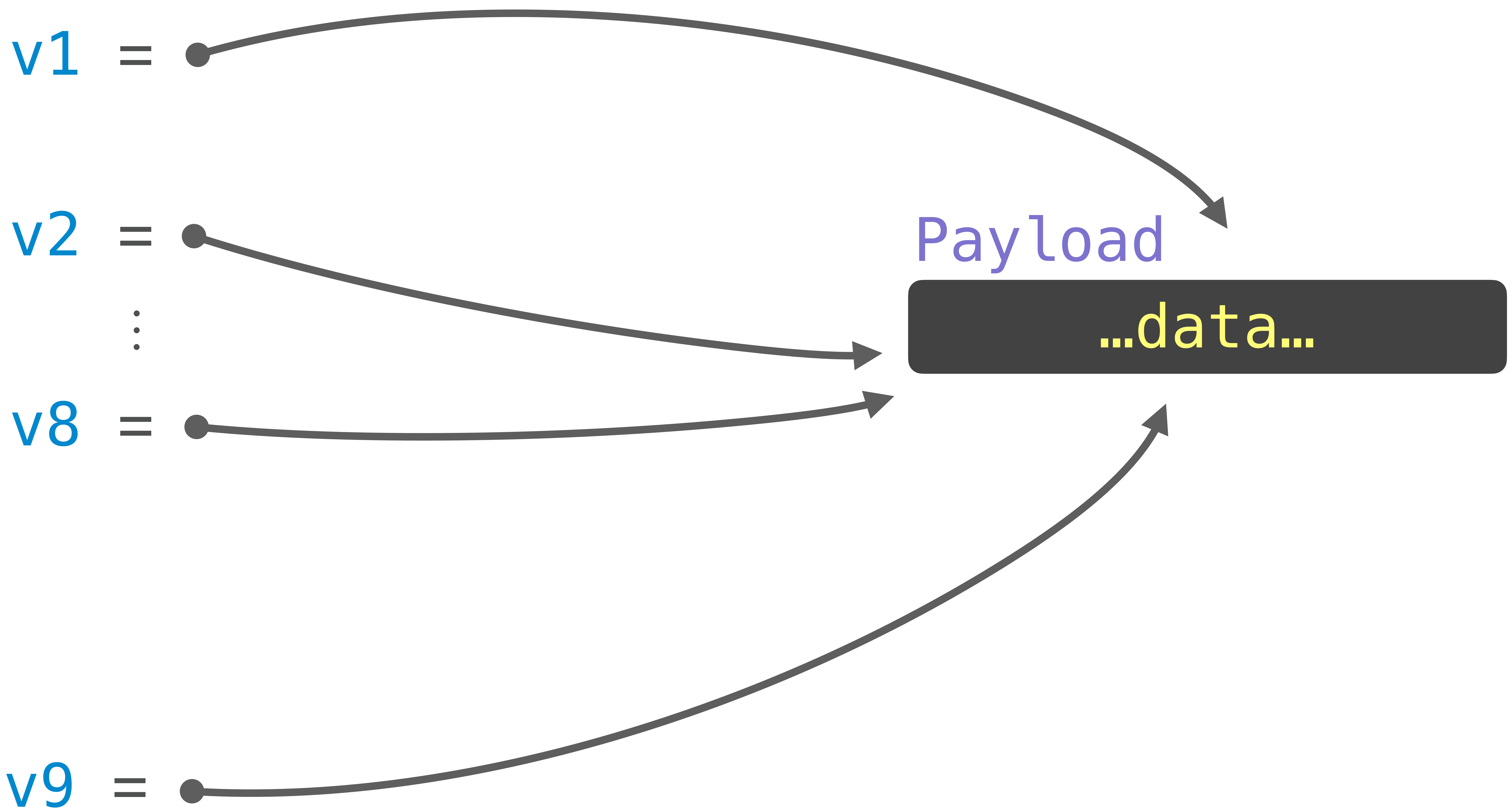
v8 =

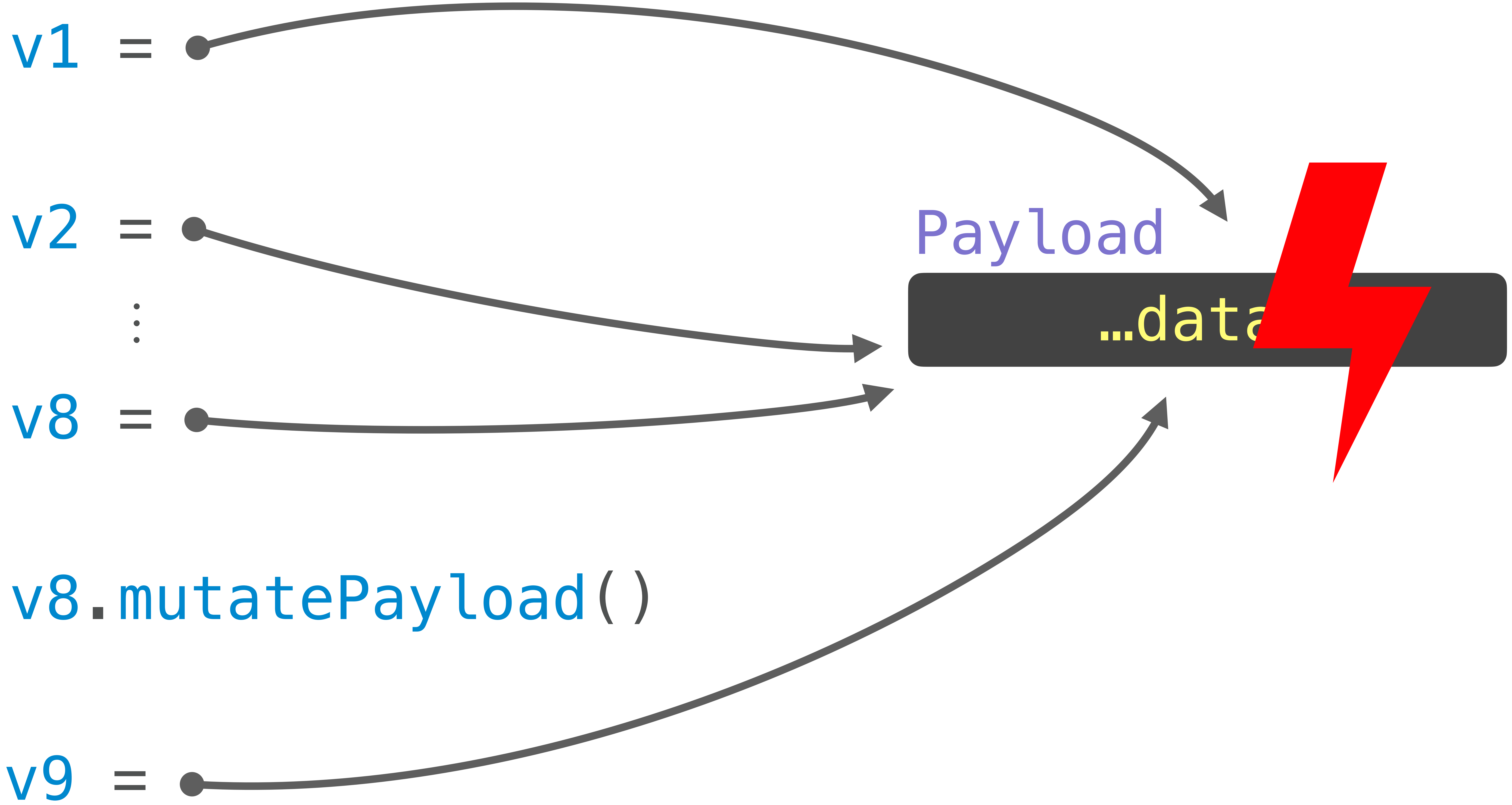
v9 =

Payload

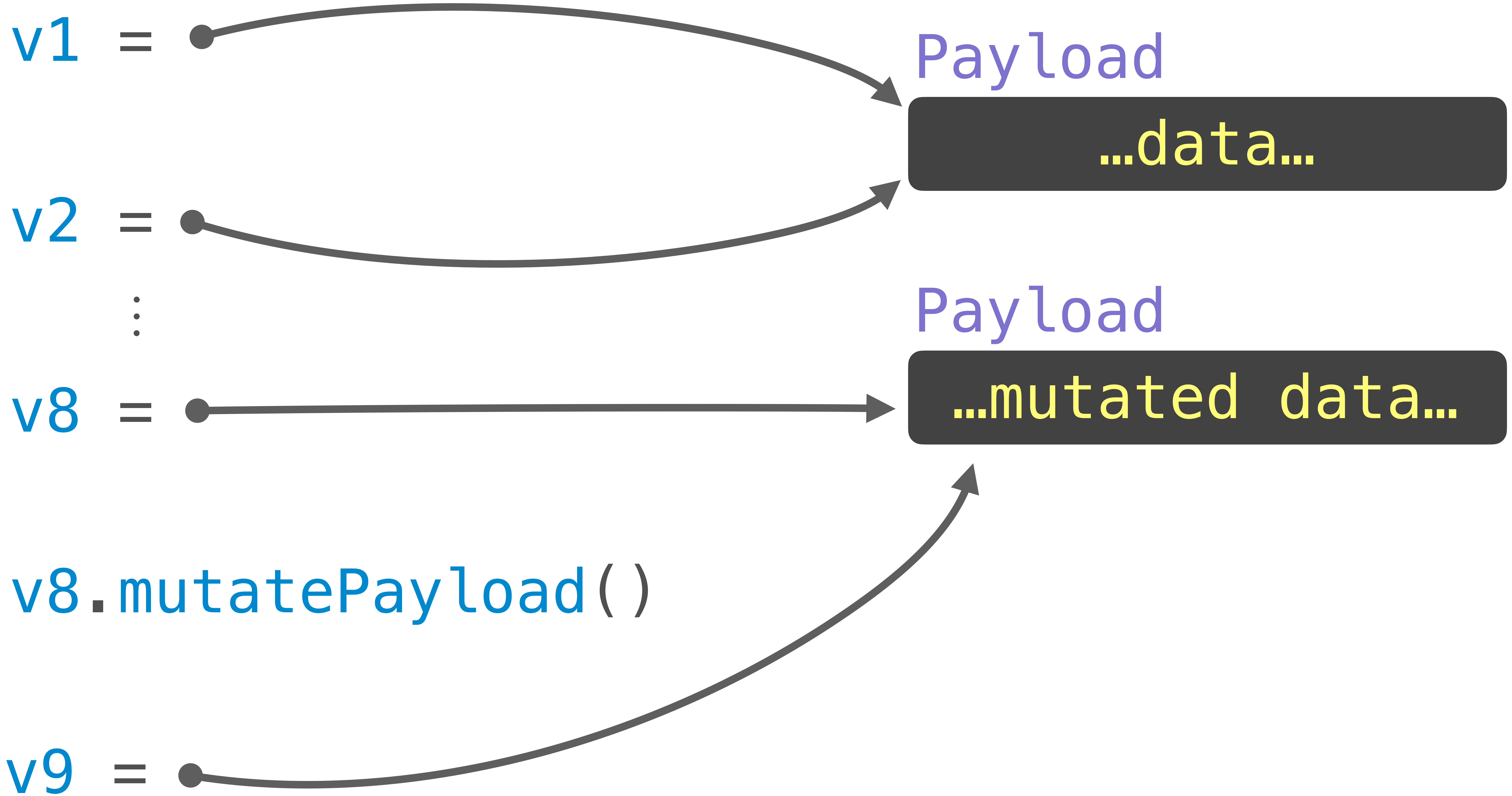
…data…

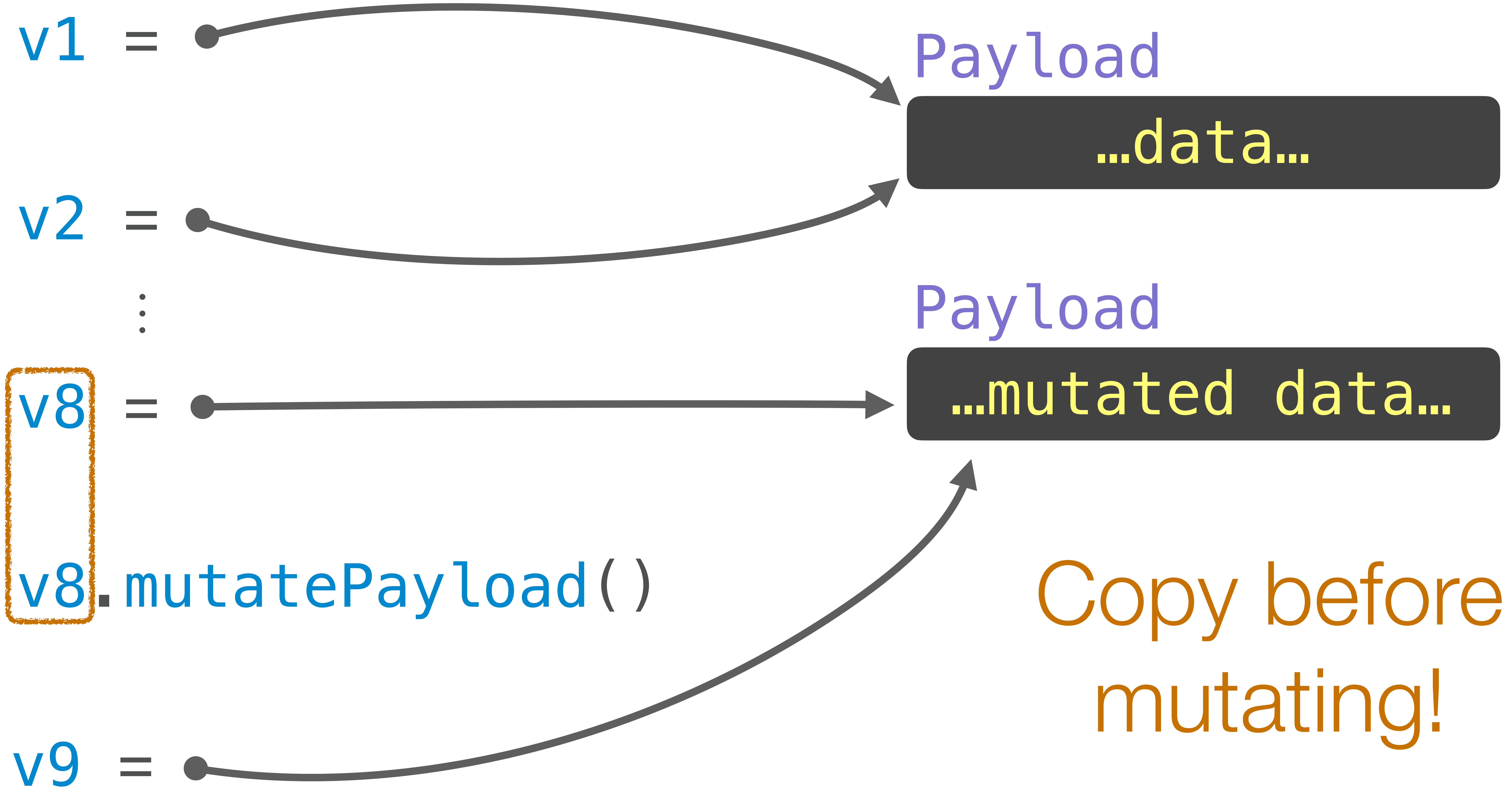

v1 =
v2 =
v8 =
v9 =
The Box
Payload
…data…


v1 =
v2 =
⋮
v8 =
v9 =
Payload
…data…



v1 =
v2 =
⋮
v8 =
v8.mutatePayload()
v9 =
Payload
…data


v1 =
v2 =
⋮
v8 =
v8.mutatePayload()
v9 =
Payload
…data…
Payload
…mutated data…



v1 =
v2 =
v8 =
v8.mutatePayload()
v9 =
Payload
…data…
Payload
…mutated data…
Copy before mutating!

Copy-on-write

Copy-on-write

Place the payload in a box

Copy-on-write

Place the payload in a box

Copy pointers to the box

Copy-on-write

Place the payload in a box

Copy pointers to the box

Copy the box, right before mutating it

Copy-on-write

Place the payload in a box

Copy pointers to the box

Copy the box, right before mutating it

if there is more than one pointer to it

Copy-on-write in Swift

Copy-on-write in Swift

The payload in the box can be

Copy-on-write in Swift

The payload in the box can be

fixed sized

Copy-on-write in Swift

The payload in the box can be

fixed sized

large struct

Copy-on-write in Swift

The payload in the box can be

fixed sized

large struct

intercept
property setters

Copy-on-write in Swift

The payload in the box can be

fixed sized

dynamically sized

large struct

intercept property setters

Copy-on-write in Swift

The payload in the box can be

fixed sized

large struct

intercept
property setters

dynamically sized

collection

Copy-on-write in Swift

The payload in the box can be

fixed sized	dynamically sized
large struct	collection
intercept property setters	intercept mutating funcs

Copy-on-write in Swift

The payload in the box can be

fixed sized	dynamically sized
large struct	collection
intercept property setters	intercept mutating funcs

Fixed-sized Structs

```
struct Bulk {
  var property1: Type1
   ⋮
  var property9: Type9
}
```

```
struct Bulk {
  var property1: Type1
   ⋮
  var property9: Type9
}
```

```
struct Bulk {
  var property1: Type1
   ⋮
  var property9: Type9
}
```

↓ Put the payload in a box

```
final class Box {
  var property1: Type1
   ⋮
  var property9: Type9
```

```
struct Bulk {
  var property1: Type1
   ⋮
  var property9: Type9
}
```

↓ Put the payload in a box

```
final class Box {
  var property1: Type1
   ⋮
  var property9: Type9

  func copy() -> Box { Box(property1, …, property9) }
}
```

```
struct Bulk {
```

```
final class Box {
  var property1: Type1
  ⋮
  func copy() -> Box { Box(property1, …, property9) }
```

```swift
struct Bulk {
  private var box: Box  // Payload
```

```swift
final class Box {
  var property1: Type1
  ⋮
  func copy() -> Box { Box(property1, …, property9) }
```

```swift
struct Bulk {
  private var box: Box  // Payload

  var property1: Type1 {
```

```swift
final class Box {
  var property1: Type1
  ⋮
  func copy() -> Box { Box(property1, …, property9) }
```

```swift
struct Bulk {
  private var box: Box  // Payload

  var property1: Type1 {
    get { box.property1 }
```

computed property

```swift
final class Box {
  var property1: Type1
  ⋮
  func copy() -> Box { Box(property1, …, property9) }
```

```swift
struct Bulk {
  private var box: Box  // Payload

  var property1: Type1 {
    get { box.property1 }
    set {
      box.property1 = newValue
    }
  ⋮
}
```

computed property

```swift
final class Box {
  var property1: Type1
  ⋮
  func copy() -> Box { Box(property1, …, property9) }
```

```swift
struct Bulk {
  private var box: Box  // Payload

  var property1: Type1 {
    get { box.property1 }
    set {
      if !isKnownUniquelyReferenced(&box) {
        box = box.copy()
      }
      box.property1 = newValue
    }
  }
  ⋮
```

computed property

```swift
final class Box {
  var property1: Type1
  ⋮
  func copy() -> Box { Box(property1, …, property9) }
```

```swift
struct Bulk {
  private var box: Box  // Payload

  var property1: Type1 {
    get { box.property1 }
    set {
      if !isKnownUniquelyReferenced(&box) {
        box = box.copy()
      }
      box.property1 = newValue
    }
  }
  ⋮
```

computed property

```swift
final class Box {
  var property1: Type1
  ⋮
  func copy() -> Box { Box(property1, …, property9) }
```

Function

isKnownUniquelyReferenced(_:)

Returns a Boolean value indicating whether the given object is known to have a single strong reference.

Declaration

```
func isKnownUniquelyReferenced<T>(_ object: inout T) -> Bool where T : AnyObject
```

Function

isKnownUniquelyReferenced(_:)

Returns a Boolean value indicating whether the given object is known to have a single strong reference.

Declaration

```
func isKnownUniquelyReferenced<T>(_ object: inout T) -> Bool where T : AnyObject
```

Swift uses reference counting

Function

isKnownUniquelyReferenced(_:)

Returns a Boolean value indicating whether the given object is known to have a single strong reference.

Declaration

```
func isKnownUniquelyReferenced<T>(_ object: inout T) -> Bool where T : AnyObject
```

Swift uses reference counting

More about this later

```swift
struct Bulk {
  private var box: Box  // Payload

  var property1: Type1 {
    get { box.property1 }
    set {
      if !isKnownUniquelyReferenced(&box) {
        box = box.copy()
      }
      box.property1 = newValue
    }
  }
  ⋮
```

```swift
final class Box {
  var property1: Type1
  ⋮
  func copy() -> Box { Box(property1, …, property9) }
```

```swift
struct Bulk {
  private var box: Box  // Payload

  var property1: Type1 {
    get { box.property1 }
    set {
      if !isKnownUniquelyReferenced(&box) {
        box = box.copy()
      }
      box.property1 = newValue
    }
  }
  ⋮
```

```swift
final class Box {
  var property1: Type1
  ⋮
  func copy() -> Box { Box(property1, …, property9) }
```

What we started with!

Automation

Plenty of boilerplate

Automation

Plenty of boilerplate

Swift macros can generate boilerplate

Automation

Plenty of boilerplate

Swift macros can generate boilerplate

https://github.com/Swift-CowBox/Swift-CowBox

"Swift-CowBox: Easy Copy-on-Write Semantics for Swift Structs"

https://www.youtube.com/watch?v=m9JZmP9E12M

Copy-on-write in Swift

The payload in the box can be

fixed sized	dynamically sized
large struct	collection
intercept property setters	intercept mutating funcs

Copy-on-write in Swift

The payload in the box can be

fixed sized	dynamically sized
large struct	collection
intercept property setters	intercept mutating funcs

Dynamically-sized Collections

```
@frozen public struct Array<Element> {
```

Dynamically-sized Collections

```
@frozen public struct Array<Element> {

  public func map<T, E>(_ transform: (Element) throws -> T)
    rethrows -> [T]
```

Dynamically-sized Collections

```
@frozen public struct Array<Element> {

  public func map<T, E>(_ transform: (Element) throws -> T)
    rethrows -> [T]

  public mutating func swapAt(_ i: Int, _ j: Int)
  ⋮
}
```

```
@frozen public struct Array<Element> {
  private var box: Box  // Payload
```

```swift
@frozen public struct Array<Element> {
  private var box: Box  // Payload

  public func map<T, E>(_ transform: (Element) throws -> T)
    rethrows -> [T]
  { box.map(transform) }
```

```swift
@frozen public struct Array<Element> {
  private var box: Box  // Payload

  public func map<T, E>(_ transform: (Element) throws -> T)
    rethrows -> [T]
  { box.map(transform) }

  public mutating func swapAt(_ i: Int, _ j: Int)
  {
```

```swift
@frozen public struct Array<Element> {
  private var box: Box  // Payload

  public func map<T, E>(_ transform: (Element) throws -> T)
    rethrows -> [T]
  { box.map(transform) }

  public mutating func swapAt(_ i: Int, _ j: Int)
  {
    if !isKnownUniquelyReferenced(&box) {
      box = box.copy()
    }
```

```swift
@frozen public struct Array<Element> {
  private var box: Box  // Payload

  public func map<T, E>(_ transform: (Element) throws -> T)
    rethrows -> [T]
  { box.map(transform) }

  public mutating func swapAt(_ i: Int, _ j: Int)
  {
    if !isKnownUniquelyReferenced(&box) {
      box = box.copy()
    }
    box.swapAt(i, j)
  }
  ⋮
}
```

Swift’s Memory Management

ARC

Memory Management

Memory Management

Swift cares about resource use

Memory Management

Swift cares about resource use

deallocation latency

Memory Management

Swift cares about resource use

deallocation latency

stop-the-world pauses

Memory Management

Swift cares about resource use

deallocation latency

stop-the-world pauses

"Swift Godot:
Fixing the Multi-million Dollar Mistake"

https://www.youtube.com/watch?v=tzt36EGKEZo

Memory Management

Swift cares about resource use

Memory Management

Swift cares about resource use

Automatic Reference Counting

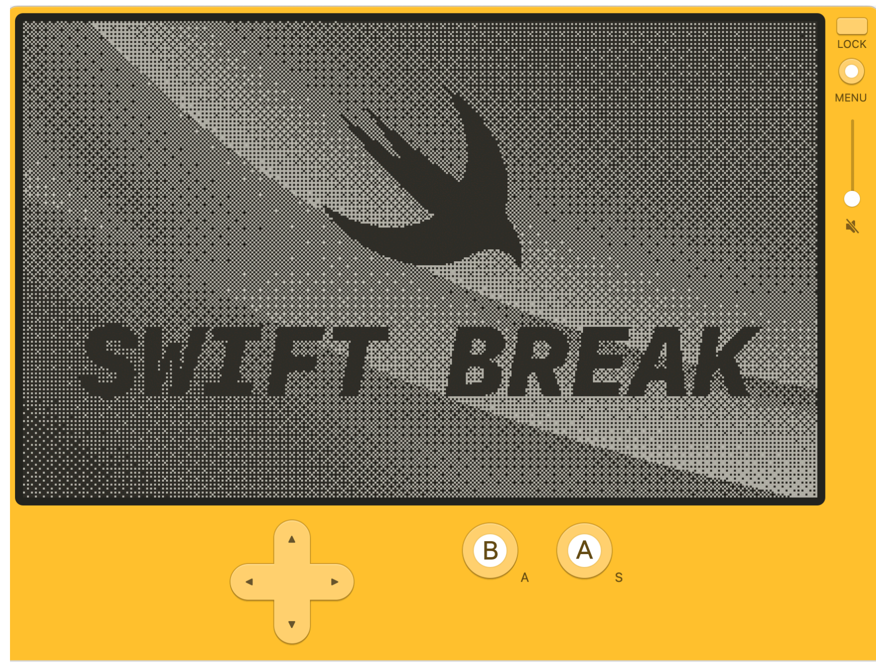
Non-copyable Types (Ownership)

Memory Management

Swift cares about resource use

Automatic Reference Counting

Non-copyable Types (Ownership)



https://www.swift.org/blog/byte-sized-swift-tiny-games-playdate/

@tacticalgrace.justtesting.org
@TacticalGrace
mchakravarty

struct enum

value types

@tacticalgrace.justtesting.org
@TacticalGrace
mchakravarty

struct enum class actor

value types reference types

struct enum class actor

value types reference types

Copy-on-write for bulk value types

@tacticalgrace.justtesting.org

@TacticalGrace

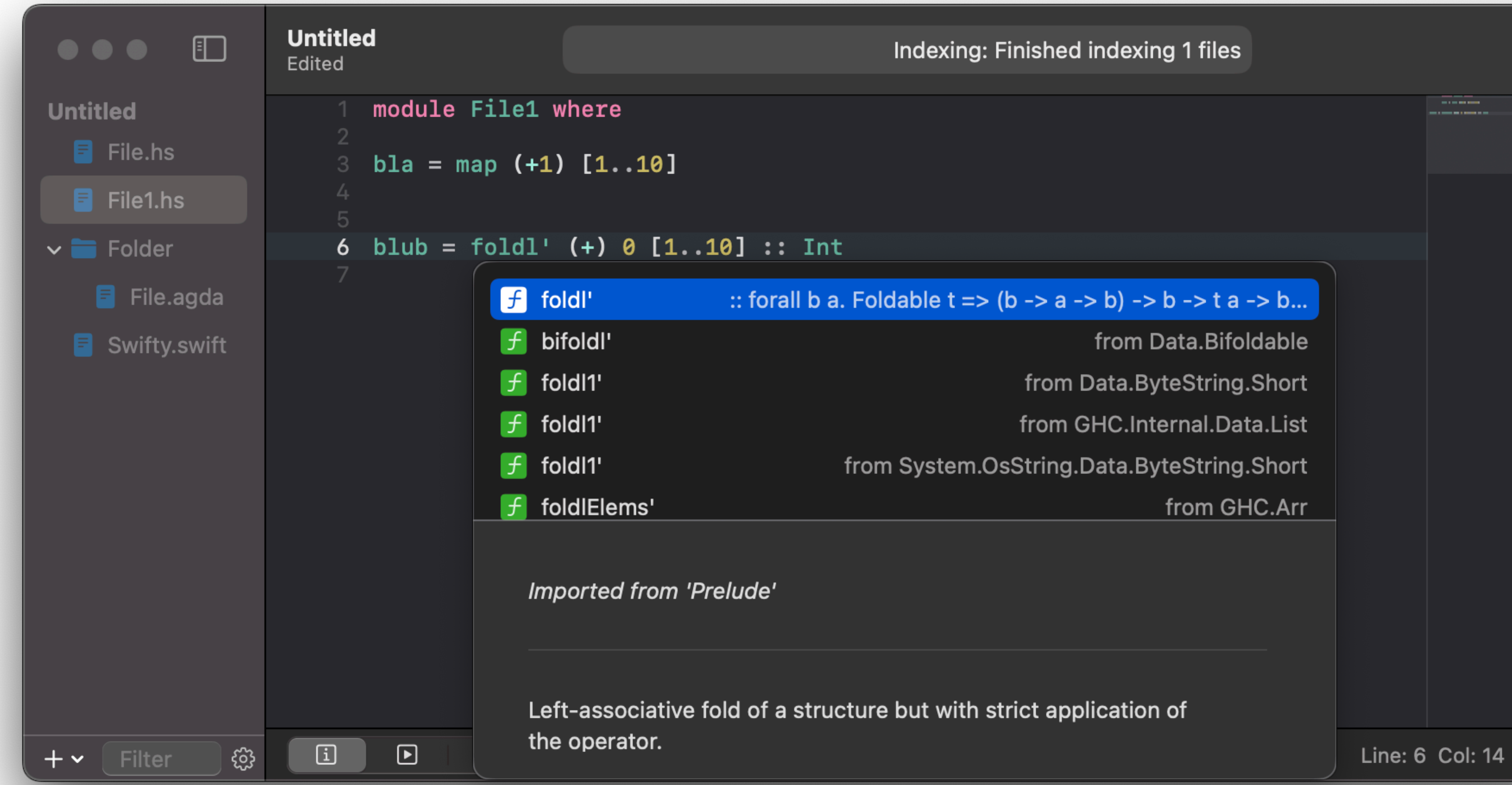
mchakravarty

struct enum class actor

value types reference types

Copy-on-write for bulk value types

Early beta testing



@tacticalgrace.justtesting.org

@TacticalGrace

mchakravarty

Thank you!

Image Attribution

https://pixabay.com/photos/stones-rocks-stack-pebbles-2040340/
https://pixabay.com/nl/photos/koe-dier-vee-zoogdier-gras-weide-6360406/

https://www.pexels.com/nl-nl/foto/vis-vissen-dierenfotografie-natuurfotografie-9004429/
https://www.pexels.com/nl-nl/foto/zwarte-vintage-camera-op-bruin-houten-plank-5211533/

Icons licensed
from Noun Project