Val Wants To Be Your Friend

DIMITRI RACORDON

Cppcon
The C++ Conference

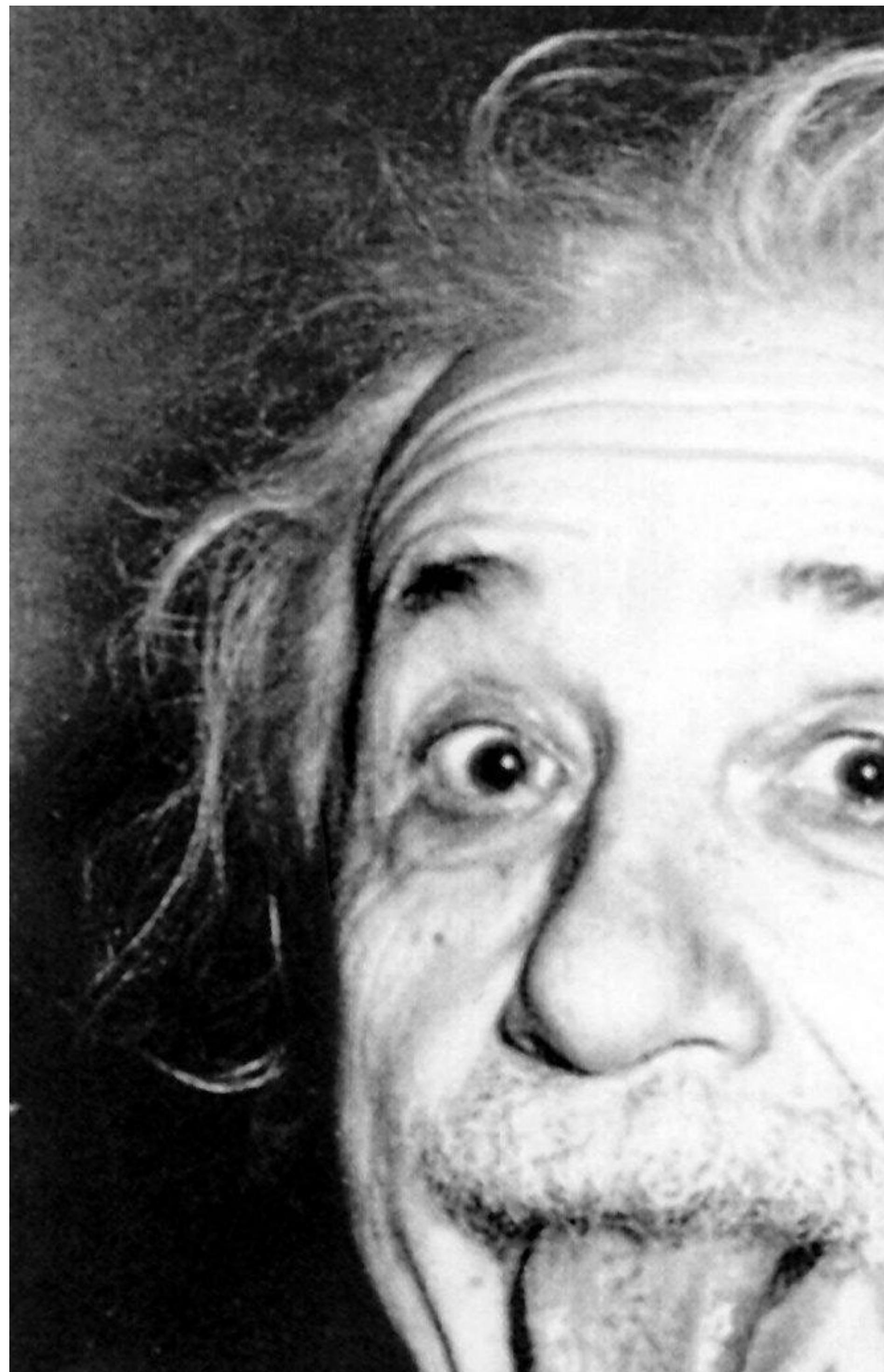
2022 | September 12th-16th

Hello, World!

```
fun main() {
  print("Hello, World!")
}
```

Mutable value semantics

> To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged.
>
> Peter O'Hearn



Mutable value semantics

```
int main() {
  std::vector v1 = { 1, 2, 3 };
  std::vector v2 = v1;
  v2[0] += 10;
  v2.push_back(4);
  std::cout << v1.size() << std::endl;
}
```

```
3
```

Mutable value semantics

```cpp
int main() {
  std::vector v1 = { 1, 2, 3 };
  std::vector v2 = v1;
  v2[0] += 10;
  v2.push_back(4);
  std::cout << v1.size() << std::endl;
}
```

3

```python
def main():
  v1 = [1, 2, 3]
  v2 = v1
  v2[0] += 10
  v2.append(4)
  print(len(v1))
```

4

shorturl.at/bckLZ

Val in a Nutshell

Can a language enforce the guarantees of MVS as practiced in C++,
without loss of efficiency?

Val in a Nutshell

What would Swift look like if it had only MVS?

What is the best way to handle non-copyable types in generic contexts?

What can MVS do for concurrency?

Val in a Nutshell

Fast by definition

Safe by default

Simple

Interoperable with C++

Val in a Nutshell

Val in a Nutshell

Val in a Nutshell

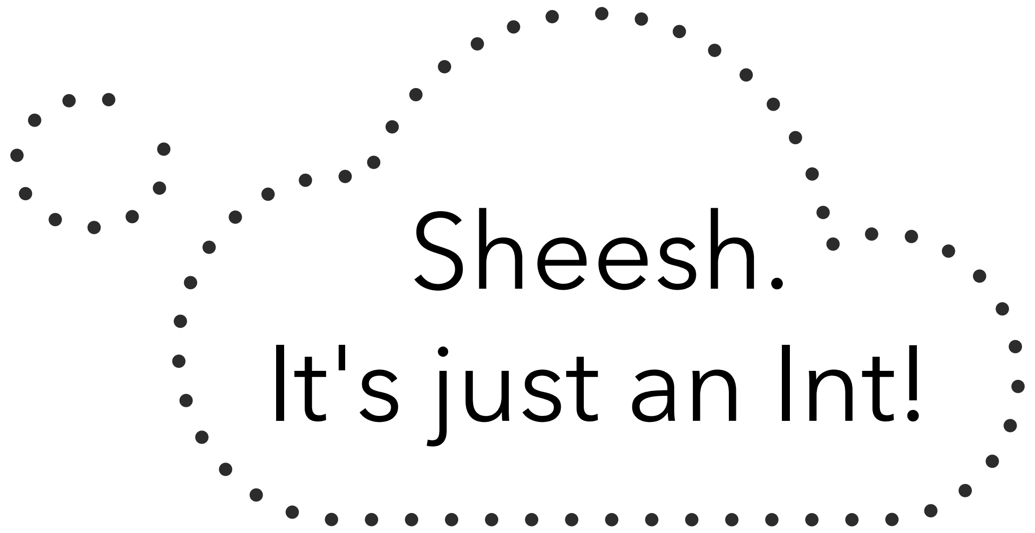
Val in a Nutshell

Explicit copies

All copies must be written explicitly in Val, yes, even copies of integers

Stay with me, it's actually not inconvenient

```
fun f(x: Int) -> Int {
  var y = x.copy()
  print(x)
  y += 1
  return y
}
```


Sheesh.
It's just an Int!

Explicit copies

```
fun print_all(_ x: Array<String>) {
  print(x.joined(by: ", "))
}

fun main() {
  var fruits = Array(["durian", "mango", "apple"])
  print_all(fruits.copy())
}
```

Explicit copies

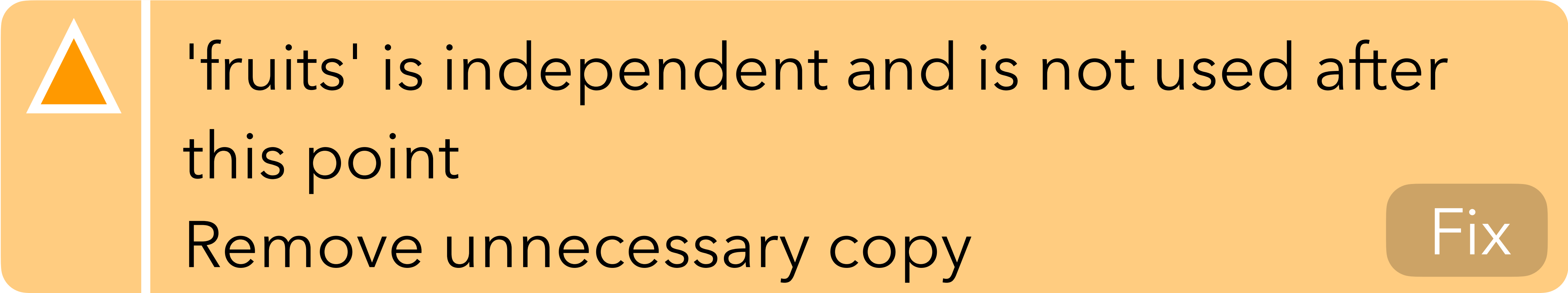
```
fun print_all(_ x: Array<String>) {
  print(x.joined(by: ", "))
}

fun main() {
  var fruits = Array(["durian", "mango", "apple"])
  print_all(fruits.copy())
}
```

Explicit copies

```
void print_all(std::vector<std::string> const& things) {
  for (auto it = things.begin(); it != things.end(); it++) {
    if (it != things.begin()) { std::cout << ", "; }
    std::cout << *it;
  }
}

int main() {
  std::vector<std::string> fruits { "durian", "mango", "apple" };
  print_count(fruits);
}
```

Explicit copies

```
fun print_all(_ things: Array<String>) {
  print(things.joined(by: ", "))
}

fun main() {
  var fruits = Array(["durian", "mango", "apple"])
  print_all(fruits.copy())
}
```



⚠ 'fruits' is independent and is not used after this point

Remove unnecessary copy

Explicit copies

```
fun unused_space(_ things: Array<String>) -> Int {
  let space = things.capacity() - things.count()
  return space.copy()
}
```

⚠ 'space' is independent and is not used after this point

Remove unnecessary copy

Fix

Passing conventions

```
type Vec2 {
  var x: Double
  var y: Double
}

fun main() {
  print(Vec2(x: 3, y: 4))
}
```

Passing conventions

```
type Vec2: Copyable {
  var x: Double
  var y: Double
}
```

Passing conventions: let

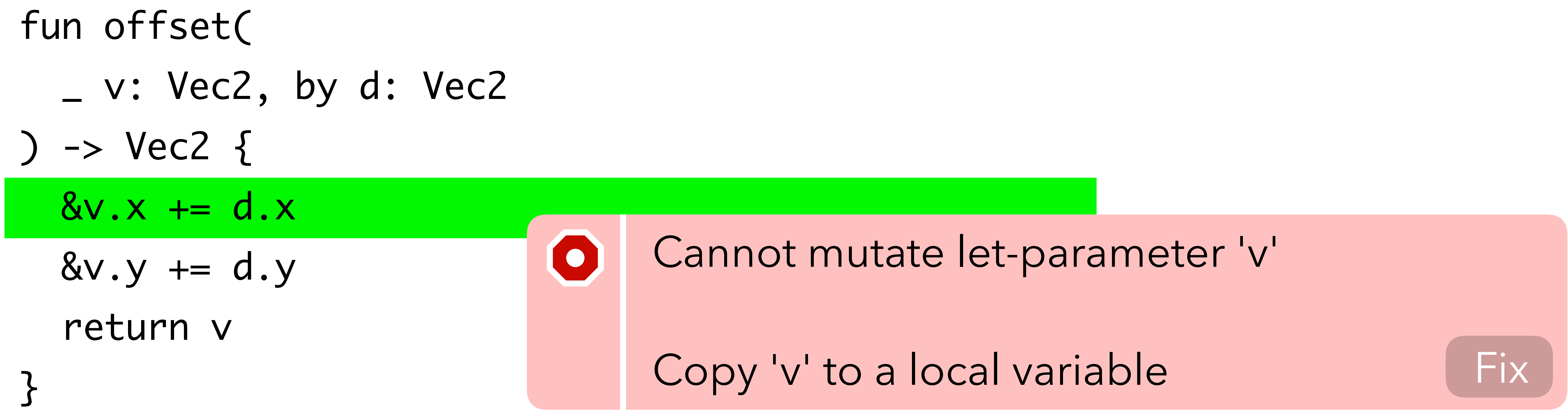
```
fun offset(
  _ v: Vec2, by d: Vec2
) -> Vec2 {
  Vec2(x: v.x + d.x, y: v.y + d.y)
}
```

Passing conventions: let

```
fun offset(
  _ v: let Vec2, by d: let Vec2
) -> Vec2 {
  Vec2(x: v.x + d.x, y: v.y + d.y)
}
```

```
auto offset(
  Vec2 const& v, Vec2 const& d
) -> Vec2 {
  return Vec2{v.x + d.x, v.y + d.y};
}
```

Passing conventions: let



```
fun offset(
  _ v: Vec2, by d: Vec2
) -> Vec2 {
  &v.x += d.x
  &v.y += d.y
  return v
}
```

Cannot mutate let-parameter 'v'

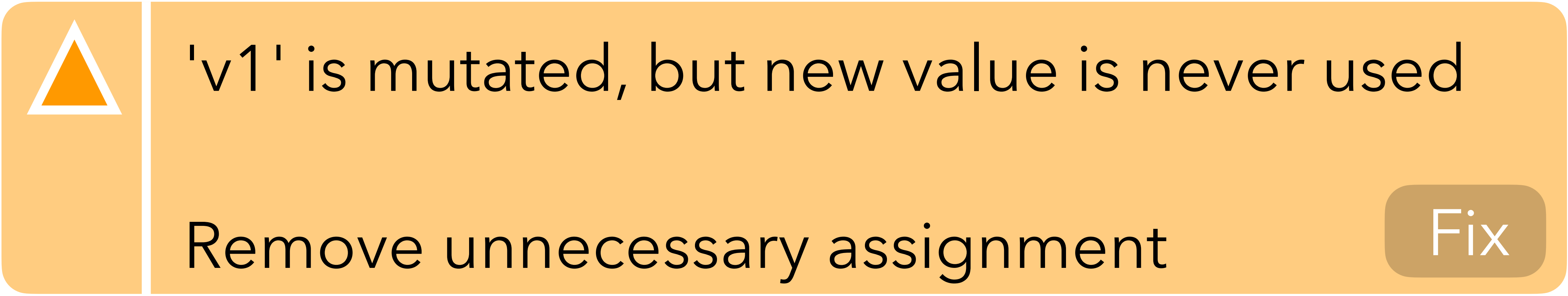
Copy 'v' to a local variable

Passing conventions: let

```
fun offset(_ v: let Vec2, by d: let Vec2) -> Vec2 {
  var _v = v.copy()
  &_v.x += d.x
  &_v.y += d.y
  return _v
}
```

Passing conventions: let

```
fun offset(_ v: let Vec2, by d: let Vec2) -> Vec2 {
  var _v = v.copy()
  &_v.x += d.x
  &_v.y += d.y
  return _v
}

fun main() {
  var v1 = Vec2(x: 1, y: 2)
  v1 = offset(v1, by: Vec2.unit_x)
  print(v1)
}
```



'v1' is mutated, but new value is never used

Remove unnecessary assignment

Passing conventions: inout

```
fun offset_inout(
  _ v: inout Vec2, by d: Vec2
) {
  &v.x += d.x
  &v.y += d.y
}
```

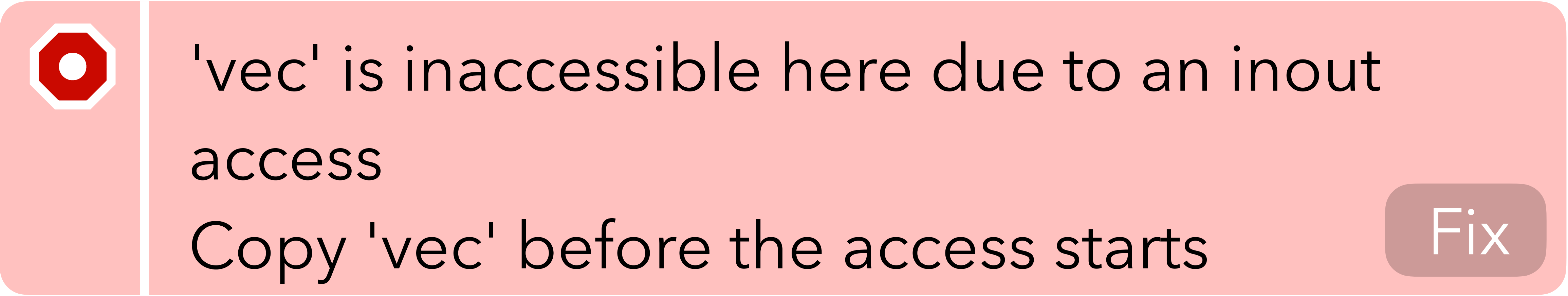
```
void offset_inout(
  Vec2& v, Vec2 const& d
) {
  v.x += d.x;
  v.y += d.y;
}
```

Passing conventions: inout

```
// Offsets `v` by `2 * d`.
void double_offset_inout(Vec2& v, Vec2 const& d) {
  offset_inout(v, d);
  offset_inout(v, d);
}

void main() {
  Vec2 vec = {3, 4};
  double_offset_inout(vec, vec);
  std::cout << vec.x << std::endl; // Should print 9, but prints 12 instead.
}
```

Passing conventions: inout

```
// Offsets `v` by `2 * d`.
fun double_offset_inout(_ v: inout Vec2, by d: Vec2) {
  offset_inout(&v, by: d)
  offset_inout(&v, by: d)
}

fun main() {
  var vec = Vec2(x: 3, y: 4)
  double_offset_inout(&vec, by: vec)
  print(vec.x)
}
```



'vec' is inaccessible here due to an inout access

Copy 'vec' before the access starts

Passing conventions: inout

```
// Offsets `v` by `2 * d`.
fun double_offset_inout(_ v: inout Vec2, by d: Vec2) {
  offset_inout(&v, by: d)
  offset_inout(&v, by: d)
}

fun main() {
  var vec = Vec2(x: 3, y: 4)
  let _vec = vec.copy()
  double_offset_inout(&vec, by: _vec)
  print(vec.x)
}
```

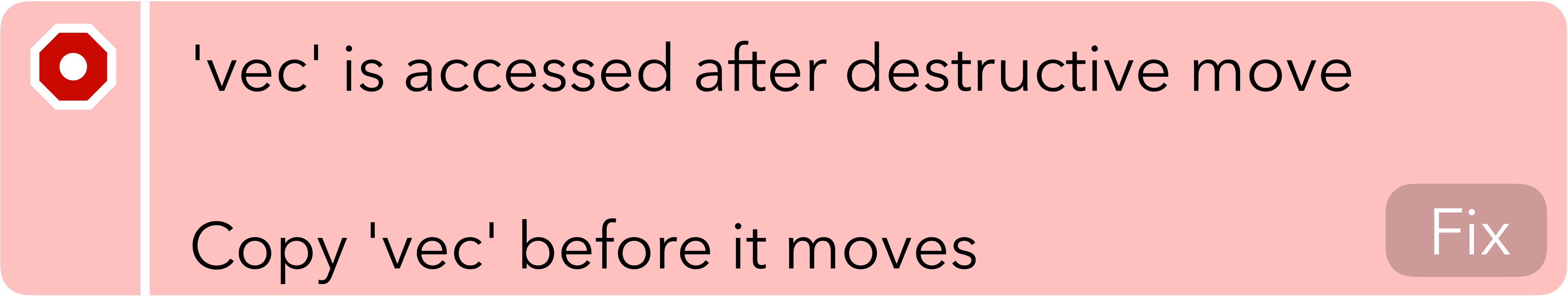
Passing conventions: sink

```
fun offset_sink(
  _ v: sink Vec2, by d: Vec2
) -> Vec2 {
  Vec2(x: v.x + d.x, y: v.y + d.y)
}

fun main() {
  var vec = Vec2(x: 3, y: 4)
  vec = offset_sink(vec, by: Vec2(x: 1, y: 1))
  print(vec.x)
}
```

```
auto offset_sink(
  Vec2 v, Vec2 const& d
) -> Vec2 {
  return Vec2{v.x + d.x, v.y + d.y};
}

int main() {
  Vec2 vec = {3, 4};
  vec = offset_sink(move(vec), {1, 1})
  std::cout << vec.x << std::endl;
}
```

Passing conventions: sink

```
fun offset_sink(
  _ v: sink Vec2, by d: Vec2
) -> Vec2 {
  Vec2(x: v.x + d.x, y: v.y + d.y)
}

fun main() {
  var vec = Vec2(x: 3, y: 4)
  vec = offset_sink(vec, by: vec)
  print(vec.x)
}
```



'vec' is accessed after destructive move

Copy 'vec' before it moves

Passing conventions: sink

```
fun offset_sink(
  _ v: sink Vec2, by d: Vec2
) -> Vec2 {
  Vec2(x: v.x + d.x, y: v.y + d.y)
}

fun main() {
  var vec = Vec2(x: 3, y: 4)
  vec = offset_sink(vec.copy(), by: vec)
  print(vec.x)
}
```

Passing conventions: sink

```
fun offset_sink(
  _ v: sink Vec2, by d: Vec2
) -> Vec2 {
  Vec2(x: v.x + d.x, y: v.y + d.y)
}

fun main() {
  defer { print("will exit main") }
  var vec = Vec2(x: 3, y: 4)
  vec = offset_sink(vec.copy(), by: vec)
  print(vec.x)
}
```

Passing conventions: sink

```
fun offset_sink(
  _ v: sink Vec2, by d: Vec2
) -> Vec2 {
  Vec2(x: v.x + d.x, y: v.y + d.y)
}

fun main() {
  defer { print("will exit main") }
  var vec = Vec2(x: 3, y: 4)
  vec = offset_sink(vec.copy(), by: vec)
  print(vec.x)
}
```

Passing conventions: inout ≡ sink

```
fun offset_sink(_ v: sink Vec2, by d: Vec2) -> Vec2 {
  offset_inout(&v, by: d)
  return v
}

fun offset_inout(_ v: inout Vec2, by d: Vec2) {
  v = offset_sink(v, by: d)
}
```

Passing conventions: sink

```
type Polygon: Copyable {
  var vertices: Array<Vec2>
}
```

```
fun offset_polygon_let(_ s: Polygon, by d: Vec2) -> Polygon { ... }
fun offset_polygon_inout(_ s: inout Polygon, by d: Vec2) { ... }
fun offset_polygon_sink(_ s: sink Polygon, by d: Vec2) -> Polygon { ... }
```

```
fun main() {
  var shape = Polygon(vertices: ...)
  shape = offset_polygon_let(shape, by: Vec2.unit_x)
  print(shape)
}
```

Passing conventions: sink

```
type Polygon: Copyable {
  var vertices: Array<Vec2>
}

fun offset_polygon_let(_ s: Polygon, by d: Vec2) -> Polygon { ... }
fun offset_polygon_inout(_ s: inout Polygon, by d: Vec2) { ... }
fun offset_polygon_sink(_ s: sink Polygon, by d: Vec2) -> Polygon { ... }

fun main() {
  var shape = Polygon(vertices: ...)
  offset_polygon_inout(&shape, by: Vec2.unit_x)
  print(shape)
}
```

Passing conventions: sink

```
type Polygon: Copyable {
  var vertices: Array<Vec2>
}

fun offset_polygon_let(_ s: Polygon, by d: Vec2) -> Polygon { ... }
fun offset_polygon_inout(_ s: inout Polygon, by d: Vec2) { ... }
fun offset_polygon_sink(_ s: sink Polygon, by d: Vec2) -> Polygon { ... }

fun main() {
  var shape = Polygon(vertices: ...)
  shape = offset_polygon_do_the_right_thing(shape, by: Vec2.unit_x)
  print(shape)
}
```

Method bundles

```
type Polygon: Copyable {
  var vertices: Array<Vec2>
  fun offset(by d: Vec2) -> Polygon {
    let   { ... }
    inout { ... }
    sink  { ... }
  }
}

fun main() {
  var shape = Polygon(vertices: ...)
  &shape.offset(by: Vec2.unit_x)
  print(shape)
}
```

Method bundles

```
type Polygon: Copyable {
  var vertices: Array<Vec2>
  fun offset(by d: Vec2) -> Polygon {
    let   { ... }
    inout { ... }
    sink  { ... }
  }
}

fun main() {
  var shape = Polygon(vertices: ...)
  print(shape.offset(by: Vec2.unit_x))
  print(shape)
}
```

Method bundles

```
type Polygon: Copyable {
  var vertices: Array<Vec2>
  fun offset(by d: Vec2) -> Polygon {
    let   { ... }
    inout { ... }
    sink  { ... }
  }
}

fun main() {
  var shape = Polygon(vertices: ...)
  print(shape.offset(by: Vec2.unit_x))

}
```

Method bundles

```
type Polygon: Copyable {
  var vertices: Array<Vec2>
  fun offset(by d: Vec2) -> Polygon {
    let   { ... }
    inout { ... }
    sink  { ... }
  }
}

fun main() {
  var shape = Polygon(vertices: ...)
  print(shape.offset(by: Vec2.unit_x))

}
```

Method bundles

```
type Polygon: Copyable {
  var vertices: Array<Vec2>
  fun offset(by d: Vec2) -> Polygon {
    let   { ... }
    // inout { self = offset.sink(by: d) }
    sink  { ... }
  }
}

fun main() {
  var shape = Polygon(vertices: ...)
  print(shape.offset(by: Vec2.unit_x))

}
```

Method bundles

```
type Polygon: Copyable {
  var vertices: Array<Vec2>
  fun offset(by d: Vec2) -> Polygon {
    let   { ... }
    // inout { self = offset.sink(by: d) }
    // sink  { offset.let(by: d) }
  }
}

fun main() {
  var shape = Polygon(vertices: ...)
  print(shape.offset(by: Vec2.unit_x))

}
```

Passing conventions: set

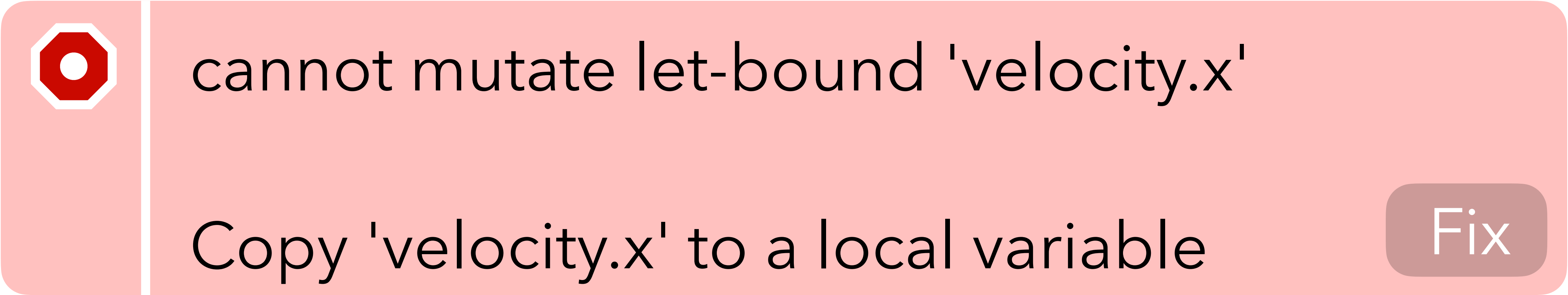
```
fun init_vector(
  _ v: set Vec2, x: sink Double, y: sink Double
) {
  v = Vec2(x: x, y: y)
}

fun main() {
  var v: Vec2
  init_vector(&v, x: 1.5, y: 2.5)
  print(v)
}
```

Local bindings

```
fun main() {
  var velocity = Vec2(x: 3, y: 4)
  let x = velocity.x
  print(x)
  &velocity.x += 1
  print(velocity)
}
```

Local bindings

```
fun main() {
  var velocity = Vec2(x: 3, y: 4)
  let x = velocity.x
  &velocity.x += 1
  print(x)
  print(velocity)
}
```



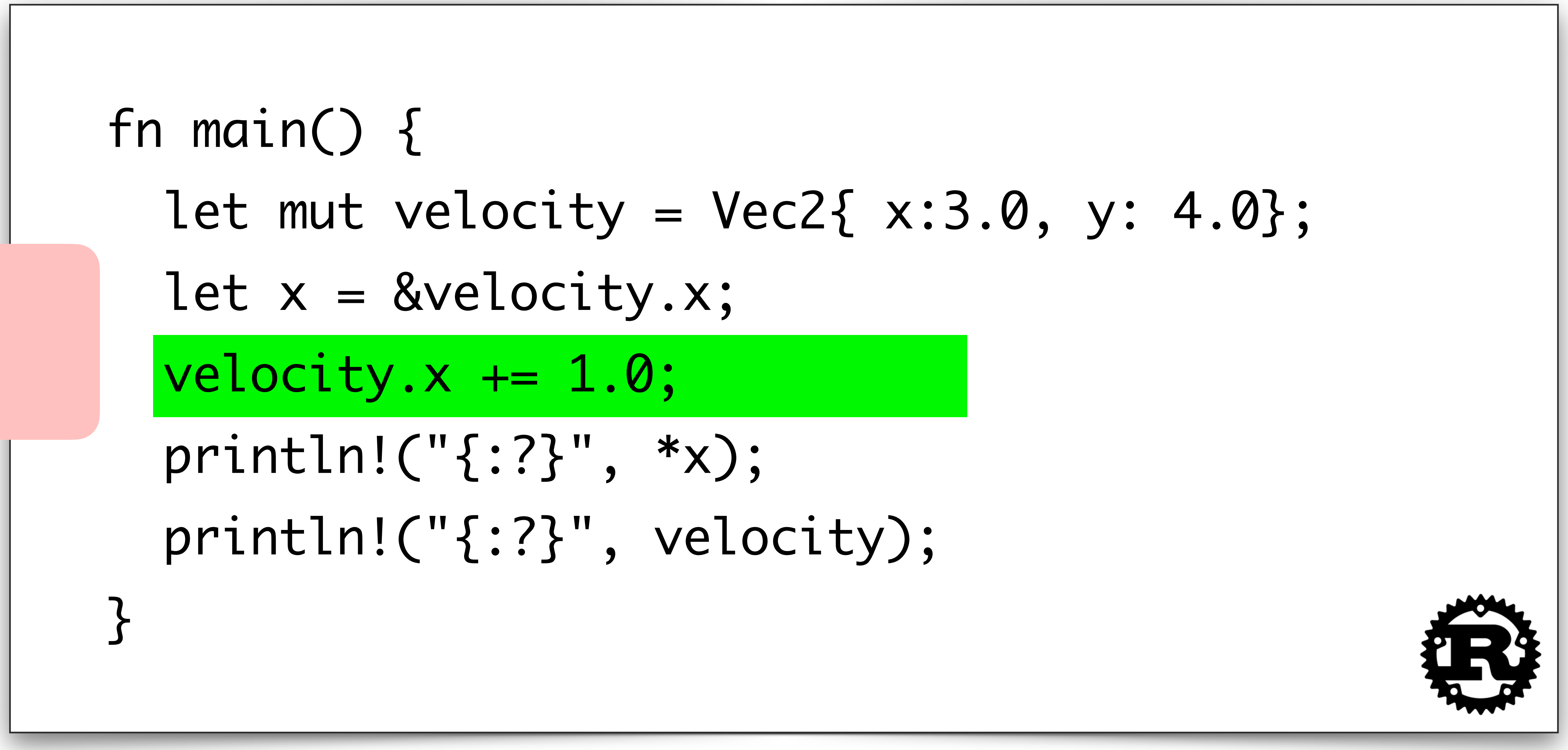
cannot mutate let-bound 'velocity.x'

Copy 'velocity.x' to a local variable

Local bindings

```
fun main() {
  var velocity = Vec2(x: 3, y: 4)
  let x = velocity.x
  &velocity.x += 1
  print(x)
  print(velocity)
}
```

cannot assign to `velocity.x` because it is borrowed



```
fn main() {
  let mut velocity = Vec2{ x:3.0, y: 4.0};
  let x = &velocity.x;
  velocity.x += 1.0;
  println!("{:?}", *x);
  println!("{:?}", velocity);
}
```

Local bindings

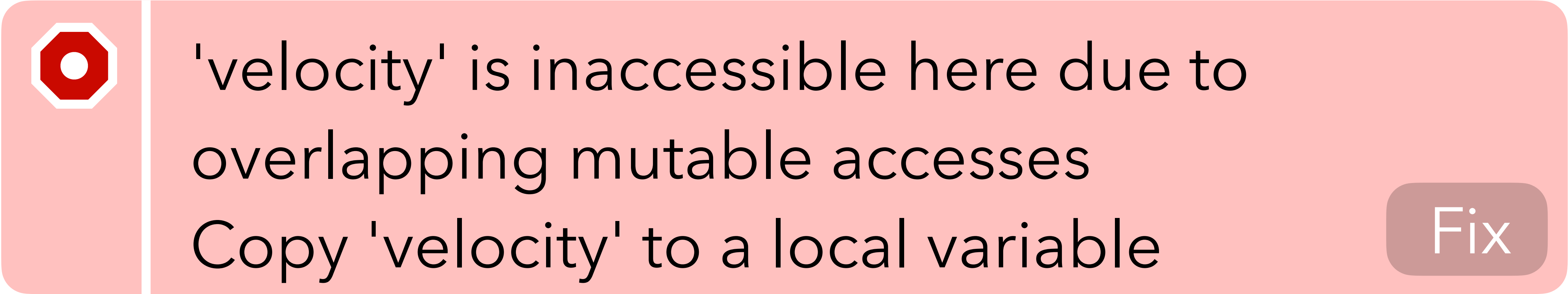
```
fun main() {
  var velocity = Vec2(x: 3, y: 4)
  let x = velocity.x
  &velocity.x += 1
  print(x)
  print(velocity)
}
```

Local bindings

```
fun main() {
  var velocity = Vec2(x: 3, y: 4)
  let x = velocity.x
  print(x)
  &velocity.x += 1
  print(velocity)
}
```

Local bindings

```
fun main() {
  var velocity = Vec2(x: 3, y: 4)
  inout x = &velocity.x
  print(velocity)
  &x += 1
  print(velocity)
}
```

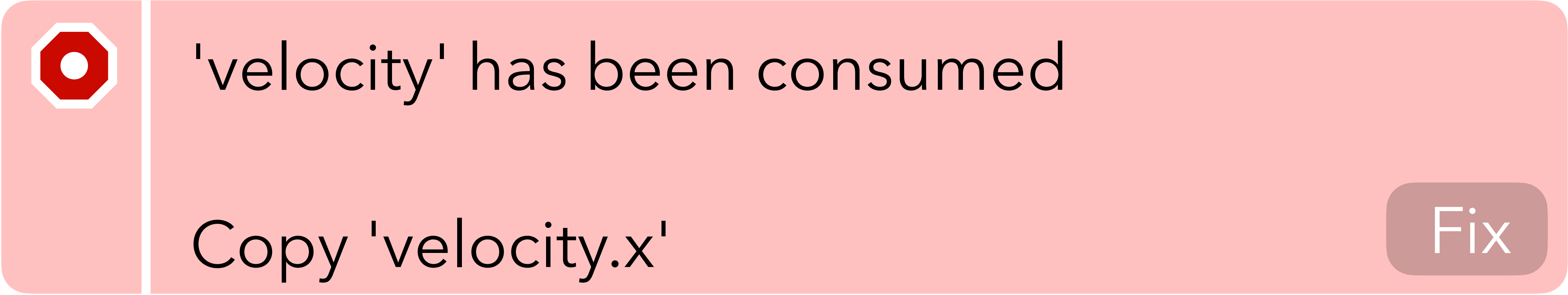


'velocity' is inaccessible here due to overlapping mutable accesses

Copy 'velocity' to a local variable

Local bindings

```
fun main() {
  var velocity = Vec2(x: 3, y: 4)
  var x = velocity.x
  print(velocity)
  &x += 1
  print(x)
}
```



'velocity' has been consumed

Copy 'velocity.x'

Projections

```
type Angle {
  var radians: Double
}

fun main() {
  var theta = Angle(radians: Double.pi)
  inout x = &theta.radians
  &x += Double.pi * 0.5
  print(theta)
}
```

Projections

```
type Angle {
  var radians: Double

  property degrees: Double {
    inout {
      var d = radians * 180.0 / Double.pi
      yield &d
      radians = d * Double.pi / 180.0
    }
  }
}
```

```
fun main() {
  var theta = Angle(radians: Double.pi)
  inout x = &theta.degrees
  &x += 90
  print(theta)
}
```

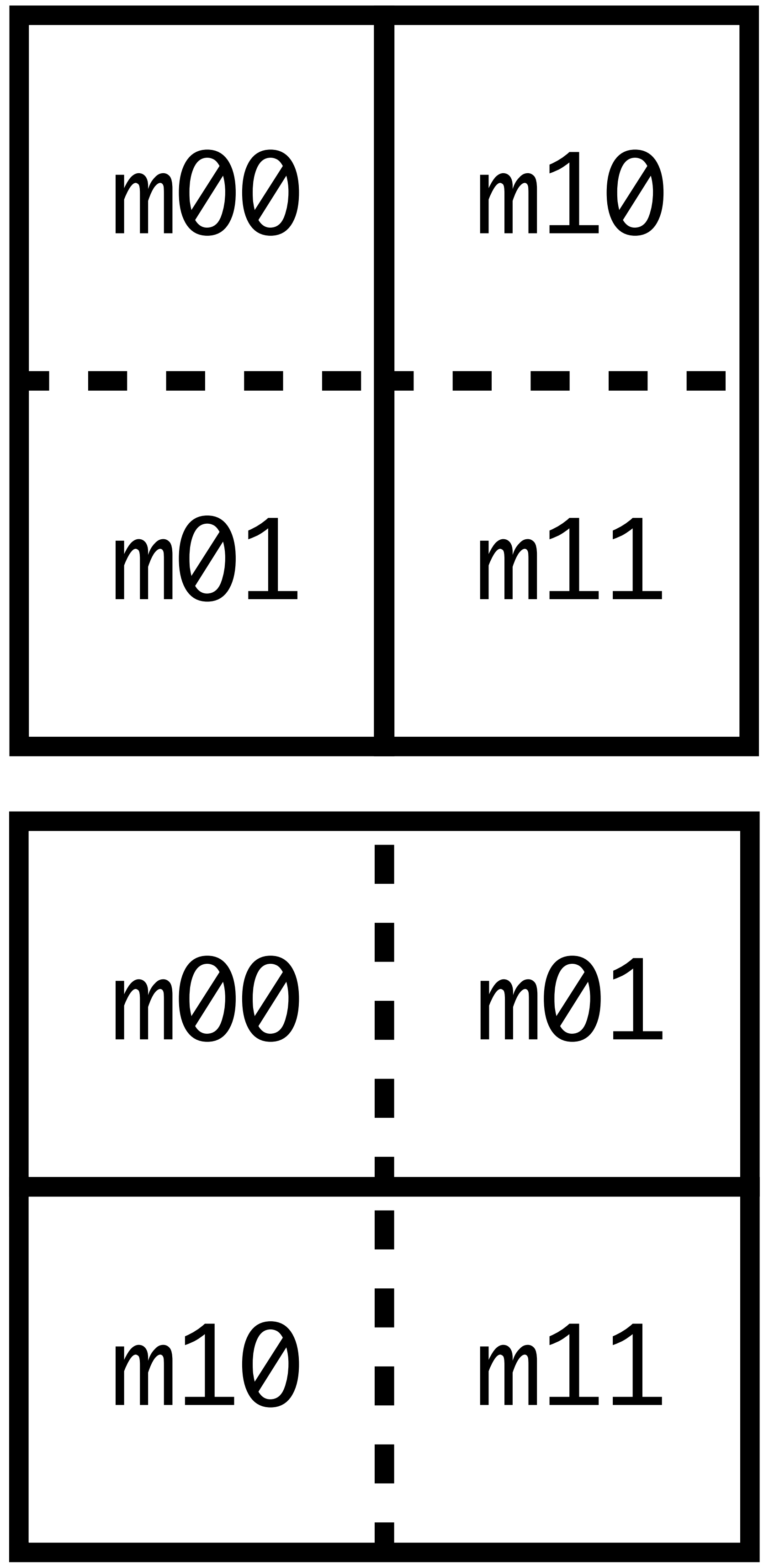
Projections

```
type Angle {
  var radians: Double

  property degrees: Double {
    inout {
      var d = radians * 180.0 / Double.pi
      yield &d
      radians = d * Double.pi / 180.0
    }
  }
}
```

```
fun main() {
  var theta = Angle(radians: Double.pi)
  inout x = &theta.degrees
  &x += 90
  print(theta)
}
```

Projections

```
type Matrix2 {
  var columns: Buffer<Buffer<Double, 2>, 2>
  subscript row(_ i: Int): Buffer<Double, 2> {










  }
}
```



Projections

```
type Matrix2 {
  var columns: Buffer<Buffer<Double, 2>, 2>
  subscript row(_ i: Int): Buffer<Double, 2> {
    let { yield [columns[0][i], columns[1][i]] }
  }
}
```

```
fun main() {
  var m = Matrix2(...)
  print(m.rows[1])
}
```

Projections

```
type Matrix2 {
  var columns: Buffer<Buffer<Double, 2>, 2>
  subscript row(_ i: Int): Buffer<Double, 2> {
    let { yield [columns[0][i], columns[1][i]] }
    inout {
      var row = [columns[0][i], columns[1][i]]
      yield &row
      columns[0][i] = row[0]; columns[1][i] = row[0]
    }


  }
}
```

```
fun main() {
  var m = Matrix2(...)
  &m.rows[1] += 1.0
}
```

Projections

```
type Matrix2 {
  var columns: Buffer<Buffer<Double, 2>, 2>
  subscript row(_ i: Int): Buffer<Double, 2> {
    let { yield [columns[0][i], columns[1][i]] }
    inout {
      var row = [columns[0][i], columns[1][i]]
      yield &row
      columns[0][i] = row[0]; columns[1][i] = row[0]
    }
    set(new_value) { columns[0][i] = new_value[0]; columns[1][i] = new_value[0] }

  }
}
```

```
fun main() {
  var m = Matrix2(...)
  m.rows[1] = [4.0, 2.0]
}
```

Projections

```
type Matrix2 {
  var columns: Buffer<Buffer<Double, 2>, 2>
  subscript row(_ i: Int): Buffer<Double, 2> {
    let { yield [columns[0][i], columns[1][i]] }
    inout {
      var row = [columns[0][i], columns[1][i]]
      yield &row
      columns[0][i] = row[0]; columns[1][i] = row[0]
    }
    set(new_value) { columns[0][i] = new_value[0]; columns[1][i] = new_value[0] }
    sink { return [columns[0][i], columns[1][i]] }
  }
}
```

```
fun main() {
  var m = Matrix2(...)
  var r = m.rows[1]
}
```

Unsafe operations

```
public type Bytes {
  var repr: {count: Int, contents: Int8[7] | MutablePointer<Int8>}

  public init(bytes: Array<Int8>) {
    if bytes.count() <= 7 {
      repr = (count: bytes.count(), contents: Int8[7](contents_of: bytes, filling_with: 0))
    } else {
      let buffer = MutablePointer.allocate(count: bytes.count())
      for i in 0 ..< bytes.count() {
        unsafe buffer.advanced(by: i).initialize(to: bytes[i].copy())
      }
      repr = (count: bytes.count(), contents: buffer)
    }
  }
```

Unsafe operations

```
public type Bytes {
  var repr: {count: Int, contents: Int8[7] | MutablePointer<Int8>}

  public init(bytes: Array<Int8>) {
    if bytes.count() <= 7 {
      repr = (count: bytes.count(), contents: Int8[7](contents_of: bytes, filling_with: 0))
    } else {
      let buffer = MutablePointer.allocate(count: bytes.count())
      for n in 0 ..< bytes.count() {
        unsafe buffer.advanced(by: n).initialize(to: bytes[n].copy())
      }
      repr = (count: bytes.count(), contents: buffer)
    }
  }
```

```
      }
      repr = (count: bytes.count(), contents: buffer)
    }
  }

  public subscript(_ i: Int): Int8 {
    inout {
      precondition(i >= 0 && i < repr.count, "index out of bounds")
      match repr.contents {
        let buffer: Int8[7] { yield &buffer[i] }
        let buffer: MutablePointer<Int8> { yield unsafe &buffer[i] }
      }
    }
  }

  public deinit { if let buffer: MutablePointer<Int8> = repr.contents { buffer.deallocate() } }
}
```

Generic programming

Alexander Stepanov explained the basics 20 years ago
https://youtu.be/1-CmNNp5eag

- Concepts are interfaces with associated types and values
- Generics are type-checked separately
- Generics use static dispatch
- But dynamic dispatch is useful too

Many languages have implemented those ideas!

Generic programming

```
/// Returns the number of occurrences of `a` in `items`.
fun occurrences<T: Collection where T.Element: Equatable>(of a: T.Element, in items) -> Int {
  items.reduce(into: 0, fun (count, e) {
    if a == e { &count += 1 }
  })
}

fun main() {
  let fruits = ["durian", "pear", "mango", "pear"]
  print(occurrences(of: "pear", in: fruits))
}
```

Generic programming

```
/// Returns the number of occurrences of `a` in `items`.
fun occurrences<T: Collection where T.Element: Equatable>(of a: T.Element, in items) -> Int {
  items.reduce(into: 0, fun (count, e) {
    if a == e { &count += 1 }
  })
}

fun main() {
  let fruits = ["durian", "pear", "mango", "pear"]
  print(occurrences(of: "pear", in: fruits))
}
```

Generic programming

```
/// Returns the number of occurrences of `a` in `items`.
fun occurrences<T: Collection where T.Element: Equatable>(of a: T.Element, in items) -> Int {
  items.reduce(into: 0, fun (count, e) {
    if a == e { &count += 1 }
  })
}
```

type 'T.Element' has no method '=='

```
fun main() {
  let fruits = ["durian", "pear", "mango", "pear"]
  print(occurrences(of: "pear", in: fruits))
}
```

Generic programming

```
/// Returns the number of occurrences of `a` in `items`.
fun occurrences<T: Collection where T.Element: Equatable>(of a: T.Element, in items) -> Int {
  items.reduce(into: 0, fun (count, e) {
    if a == e { &count += 1 }
  })
}

fun main() {
  let things = [Incomparable(), Incomparable(), Incomparable(), Incomparable()]
  print(occurrences(of: Incomparable(), in: things))
}
```

function call requires that 'Incomparable' conform to 'Equatable'

Generic programming

```
template<typename T, typename Element>
  requires Reducible<T, Element, int> && std::equality_comparable<Element>
int occurrences(Element const& a, T const& items) {
  return reduce(items, 0, std::function<void(int&, Element const&)>(
    [&a](auto& count, auto const& e) { if (a == e) { count += 1; } }));
}

int main() {
  std::vector<Incomparable> things = {{}, {}, {}, {}};
  std::cout << occurrences(Incomparable{}, things) << std::endl;
}
```

function call requires that 'Incomparable' conform to 'Equatable'

Generic programming

```
template<typename T, typename Element>
  requires Reducible<T, Element, int>
int occurrences(Element const& a, T const& items) {
  return reduce(items, 0, std::function<void(int&, Element const&)>(
    [&a](auto& count, auto const& e) { if (a == e) { count += 1; } }));
}

int main() {
  std::vector<Incomparable> things = {{}, {}, {}, {}};
  std::cout << occurrences(Incomparable{}, things) << std::endl;
}
```

Generic programming

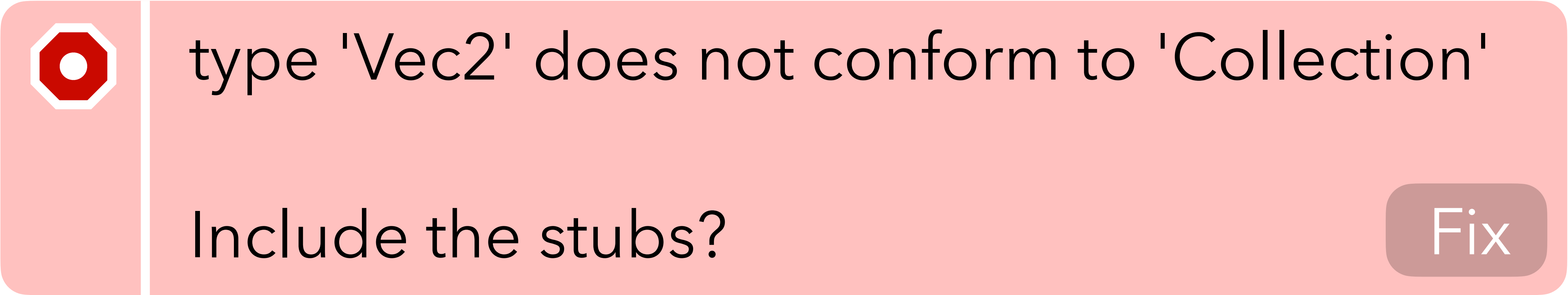
```
/// A type representing a collection of elements that can be traversed nondestructively.
trait Collection {
  type Element
  type Index: Copyable, Equatable

  fun first_index() -> Index
  fun end_index() -> Index
  fun index(after i: Index) -> Index

  subscript(_ i: Index): Index { let }

  fun is_empty() -> Bool { first_index() == end_index() }
}
```

Generic programming

```
/// A collection that can be traversed in both directions.
trait BidirectionalCollection: Collection {
  fun index(before i: Index) -> Index
}
```

Generic programming

```
type Vec2: Copyable, Collection {
  var x: Double
  var y: Double
}
```



type 'Vec2' does not conform to 'Collection'

Include the stubs?

Generic programming

```
type Vec2: Copyable {
  var x: Double
  var y: Double
}

conformance Vec2: Collection {
  typealias Element = Double
  typealias Index = Int

  fun start_index() -> Int { 0 }
  fun end_index() -> Int { 2 }
  fun index(after i: Int) -> Int { i + 1 }

  subscript(_ i: Int): Double { if i == 0 { x } else { y } }
}
```

Generic programming

```
namespace Foo {
  conformance Vec2: Collection {
    typealias Element = Double
    typealias Index = Int

    fun start_index() -> Int { 0 }
    fun end_index() -> Int { 2 }
    fun index(after i: Int) -> Int { i + 1 }

    subscript(_ i: Int): Double { if i == 0 { x } else { y } }
  }
}
```

Generic programming

```
extension Collection where Element: Copyable {
  fun reverse() -> Array<Element> {
    var result = Array(self)
    let count = result.count()
    for i in 0 ..< count / 2 {
      &result.swap_at(i, count - (i + 1))
    }
    return result
  }
}

fun main() {
  print([1, 2, 3].reversed().is_empty())
}
```

Generic programming

```
extension Collection where Element: Copyable {
  fun reverse() -> Array<Element> {
    var result = Array(self)
    let count = result.count()
    for i in 0 ..< count / 2 {
      &result.swap_at(i, count - (i + 1))
    }
    return result
  }
}

fun main() {
  print([1, 2, 3].reversed().is_empty())
}
```

Generic programming

```
extension Collection where Element: Copyable {
  fun reverse() -> some Collection where .Element == Element {
    var result = Array(self)
    let count = result.count()
    for i in 0 ..< count / 2 {
      &result.swap_at(i, count - (i + 1))
    }
    return result
  }
}

fun main() {
  print([1, 2, 3].reversed().is_empty())
}
```

Generic programming

```
extension Vec2 {
  fun reverse() -> some Collection where .Element == Element {
    Vec2(x: y.copy(), y: x.copy())
  }
}
```

Generic programming

```
type Circle {
  var center: Vec2
  var radius: Double
}

type Rectangle {
  var center: Vec2
  var dimensions: Vec2
}
```

Breaking Dependencies
Type Erasure - The Implementation Details

- Klaus Iglberger, CppCon 2022

Generic programming

```
type Canvas { ... }

trait Drawable {
  // Draws `self` onto `canvas`.
  fun draw(onto canvas: inout Canvas)
}

conformance Circle: Drawable {
  fun draw(onto canvas: inout Canvas) { ... }
}
conformance Rectangle: Drawable {
  fun draw(onto canvas: inout Canvas) { ... }
}
```

Generic programming

```
fun main() {
  var shapes_to_draw: Array<???>
  shapes.append(Circle(...))
  shapes.append(Rectangle(...))
  shapes.append(Circle(...))
}
```

Inheritance is the base class of Evil

- Sean Parent, Going Native 2013

Generic programming

```
class Drawable {
private:
  struct DrawableConcept {
    virtual ~DrawableConcept() {}
    virtual std::unique_ptr<DrawableConcept> clone() const = 0;
    virtual void draw(Canvas& canvas) const = 0;
  };

  template<typename T>
  struct DrawableModel: DrawableConcept {
    DrawableModel(T&& value): object{ std::forward<T>(value) } {}
    std::unique_ptr<DrawableConcept> clone() const override { return std::make_unique<DrawableModel>(*this); }
    void draw(Canvas& canvas) const override { draw(object, canvas); }
    T object;
  };

  friend void draw(Drawable const& drawable, Canvas& canvas) { drawable.pimpl->draw(canvas); }

  std::unique_ptr<DrawableConcept> pimpl;

public:
  template<typename T>
  Drawable(T const& x): pimpl{ new DrawableModel<T>( x ) } {}
  // ...
};
```

Generic programming

```
fun main() {
  var shapes_to_draw: Array<any Drawable>
  shapes.append(Circle(...))
  shapes.append(Rectangle(...))
  shapes.append(Circle(...))
}
```

Generic programming

```
fun main() {
  var shapes_to_draw: Array<any Drawable>
  shapes.append(Circle(...))
  shapes.append(Rectangle(...))
  shapes.append(Circle(...))

  var canvas = Canvas()
  draw_all(shapes: shapes_to_draw, onto: &canvas)
}
```

```
fun draw_all<T: Collection where T: Drawable>(shapes: T, onto canvas: inout Canvas) {
  for let s in shapes { s.draw(onto: &canvas) }
}
```

Generic programming

```
fun main() {
  var shapes_to_draw: Array<any Drawable>
  shapes.append(Circle(...))
  shapes.append(Rectangle(...))
  shapes.append(Circle(...))

  var canvas = Canvas()
  draw_all(shapes: shapes_to_draw, onto: &canvas)
}

fun draw_all<T: Collection where T: Drawable>(shapes: T, onto canvas: inout Canvas) {
  for let s in shapes { s.draw(onto: &canvas) }
}
```

Generic programming

```
fun main() {
  var fruits: any Collection where .Element == String
  fruits = if Bool.random() { Array(["mango", "pear"]) } else { LinkedList(["mango", "pear"]) }
  if let f = fruits.first {
    print(f == "durian")
  }
  print(fruits[0])
}
```

Generic programming

```
fun main() {
  var fruits: any Collection where .Element == String
  fruits = if Bool.random() { Array(["mango", "pear"]) } else { LinkedList(["mango", "pear"]) }
  if let f = fruits.first {
    print(f == "durian")
  }
  print(fruits[0])
}
```

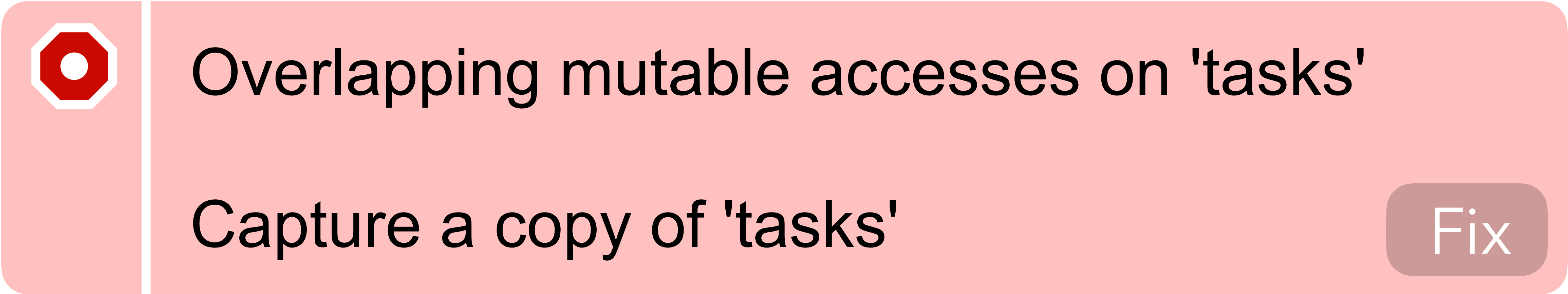
type 'Int' is not equal to unspecified existential type

Concurrency

```
fun long_task(input: Int) -> Int {
  var result = 0
  for let i in 0 ..< 42 { sleep(1); &result += 1 }
  return result
}

fun main() {
  let f1 = spawn 1 + 2
  let f2 = spawn: Int {
    sink var i = long_task(input: 0)
    return i + 1
  }
  print((join f1, f2))
}
```

Concurrency

```
fun process() {
  var tasks = get_tasks()
  let f1 = spawn { while let task = &tasks.pop_first() { task.run() } }
  let f2 = spawn { while let task = &tasks.pop_first() { task.run() } }
  _ = join f1, f2
}
```



Overlapping mutable accesses on 'tasks'

Capture a copy of 'tasks'

Concurrency

```
fun process() {
  var tasks = get_tasks()
  var (r1, r2) = tasks.sliced[at: tasks.count() / 2]
  let f1 = spawn { while let task = &r1.pop_first() { task.run() } }
  let f2 = spawn { while let task = &r2.pop_first() { task.run() } }
  _ = join f1, f2
}
```

Comparison with Carbon

Comparison with Carbon

Safety by construction vs post-hoc safety

Safety by construction vs post-hoc safety

What's best for the future of C++?

Safe is better than safer.

Can we get from safer to safe?

Comparison with C++ Core Guidelines

Safety by construction vs post-hoc safety

https://val-lang.dev

Thanks for your attention