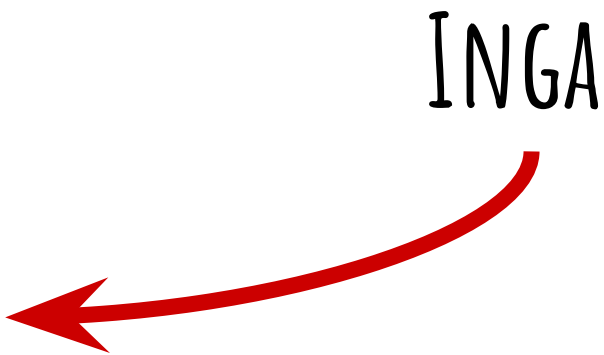
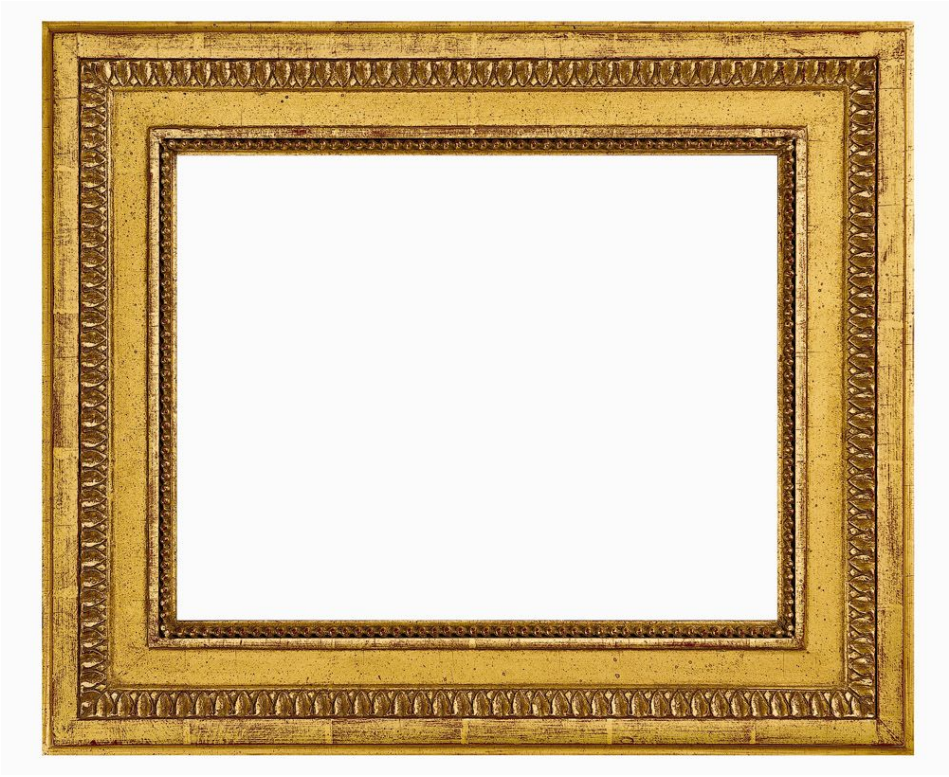

Inga

Less
Is
more

GRAIN -R = **GAIN**

POWDER -D = **POWER**



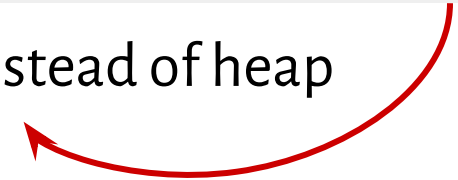
FRAME -R = **FAME**

SWIFTC -O = EFFICIENT PROGRAMS

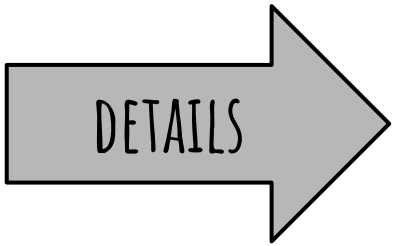
Optimize memory usage

Optimize memory usage

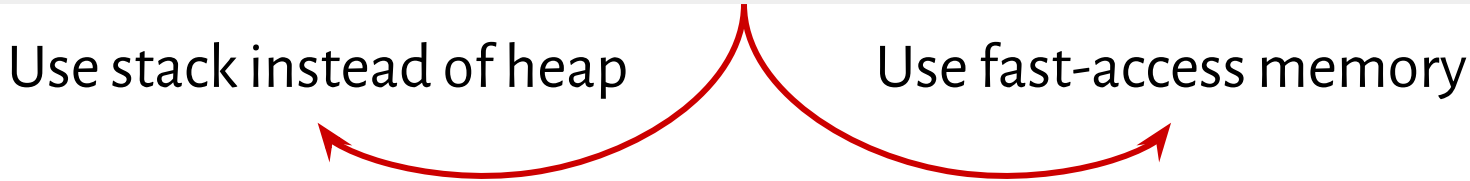
Use stack instead of heap



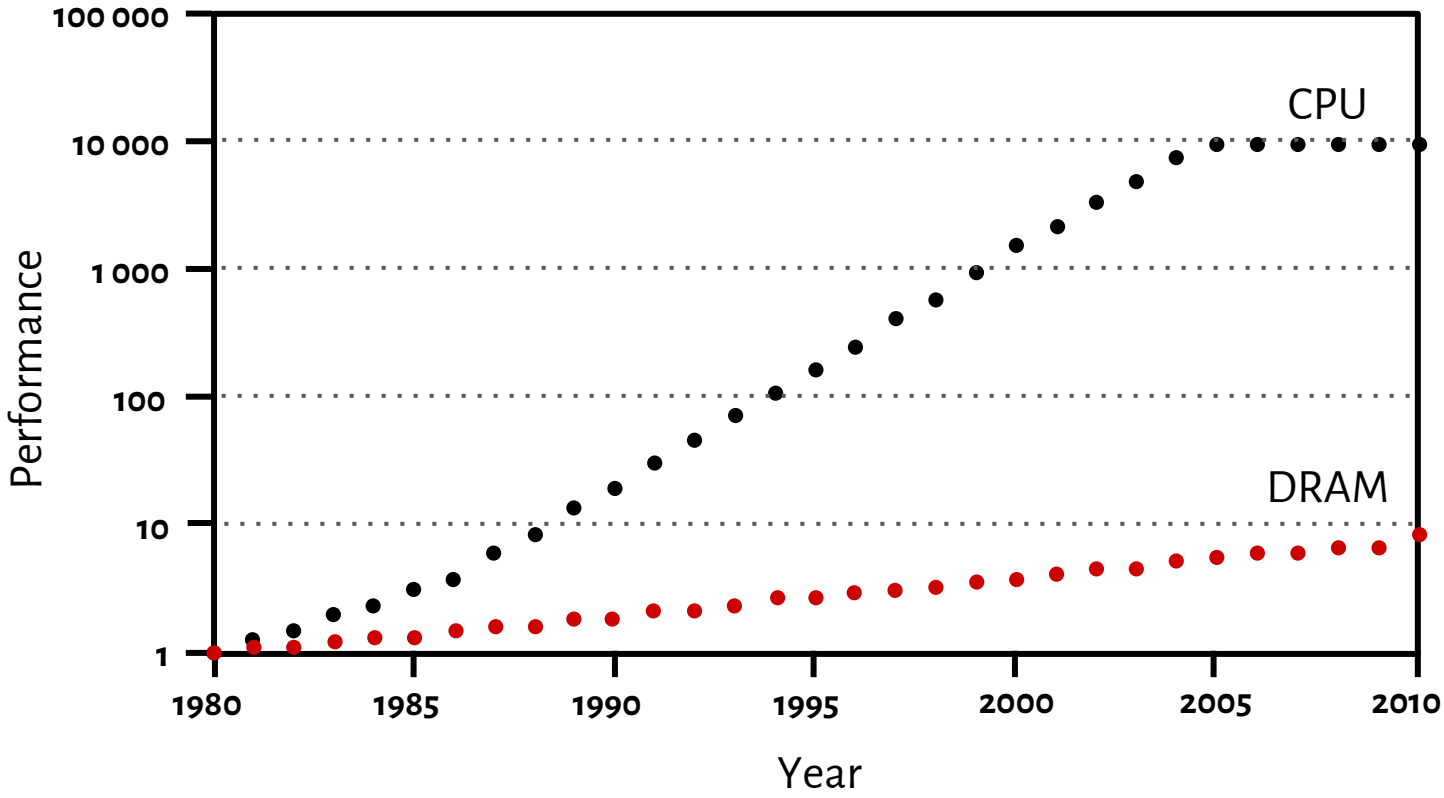
Why do we prefer stack allocations?



Optimize memory usage

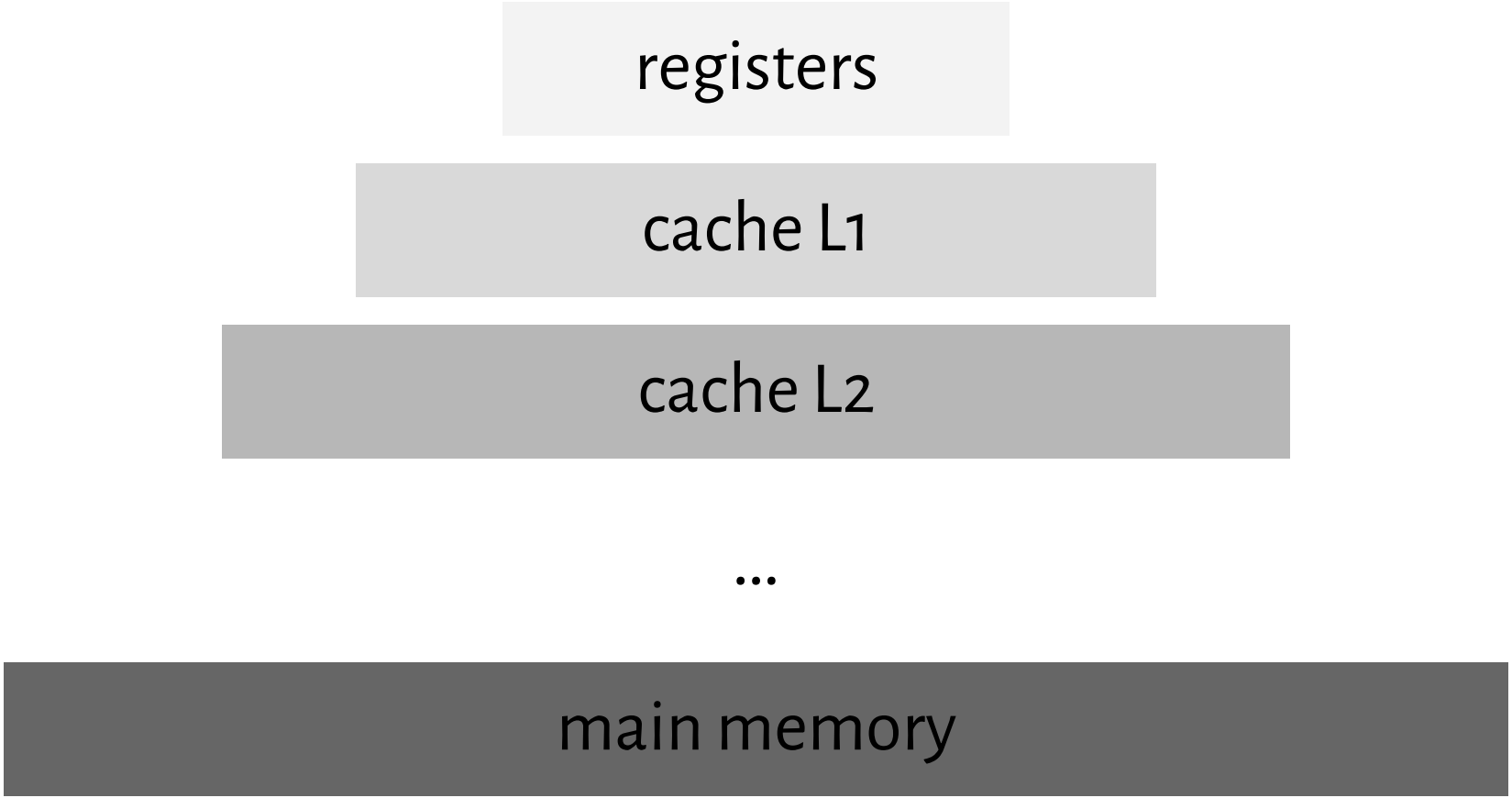


Memory-access bottleneck

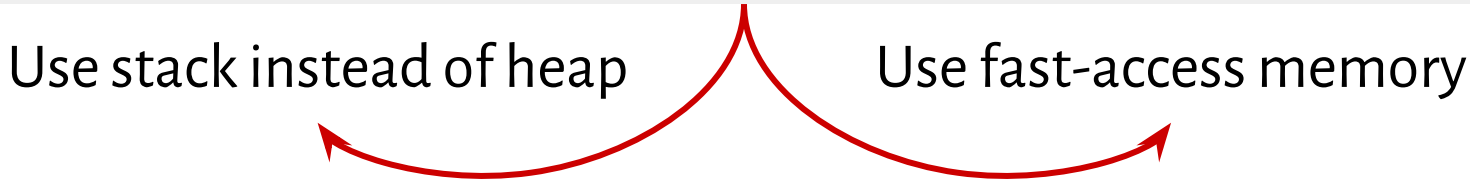


J. L. Hennessy, Stanford University and D. A. Patterson, University of California, Berkeley: *Memory Hierarchy Design*, 2012

Memory hierarchy

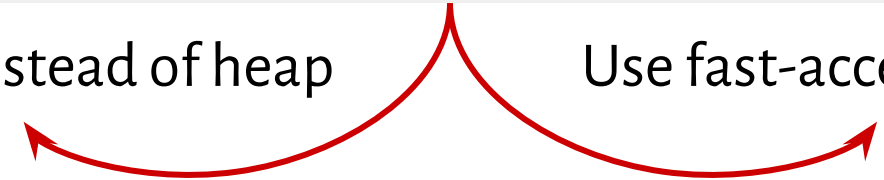


Optimize memory usage



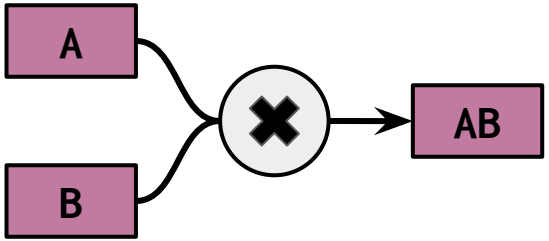
Optimize processor usage

Optimize memory usage

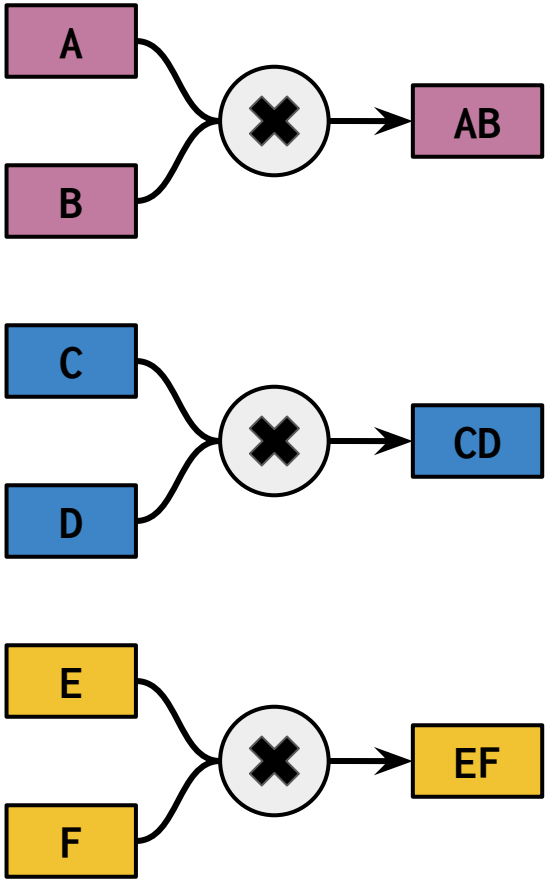


Optimize processor usage

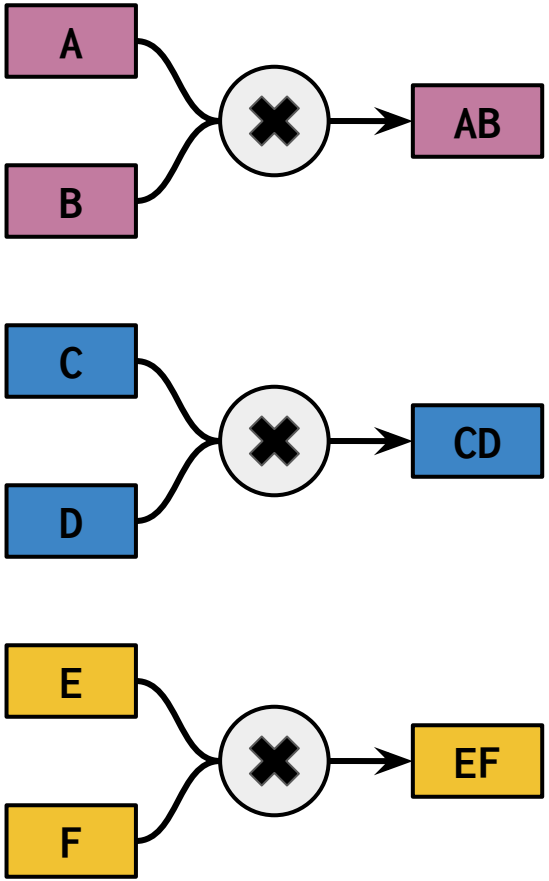
Parallelism

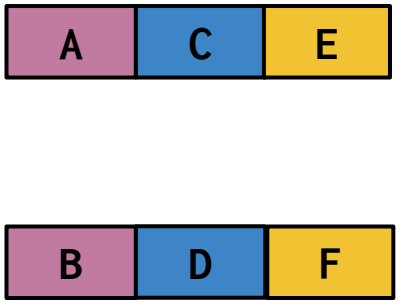


Parallelism

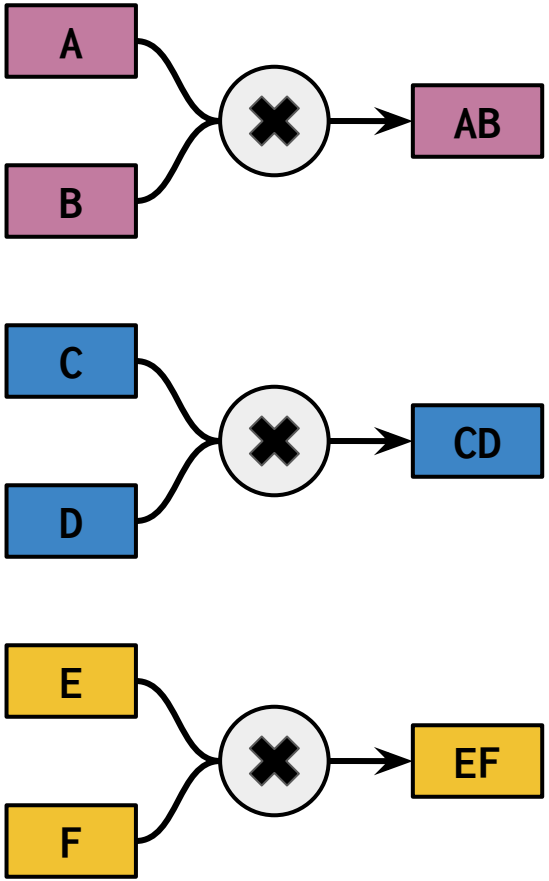


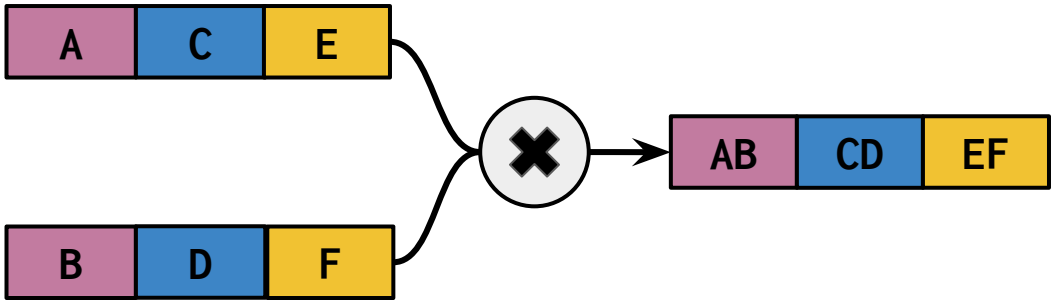
Parallelism



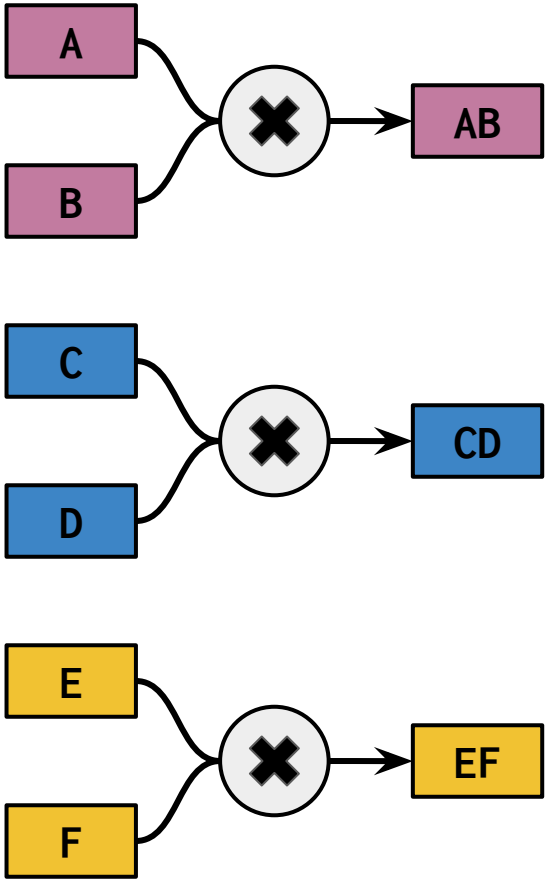


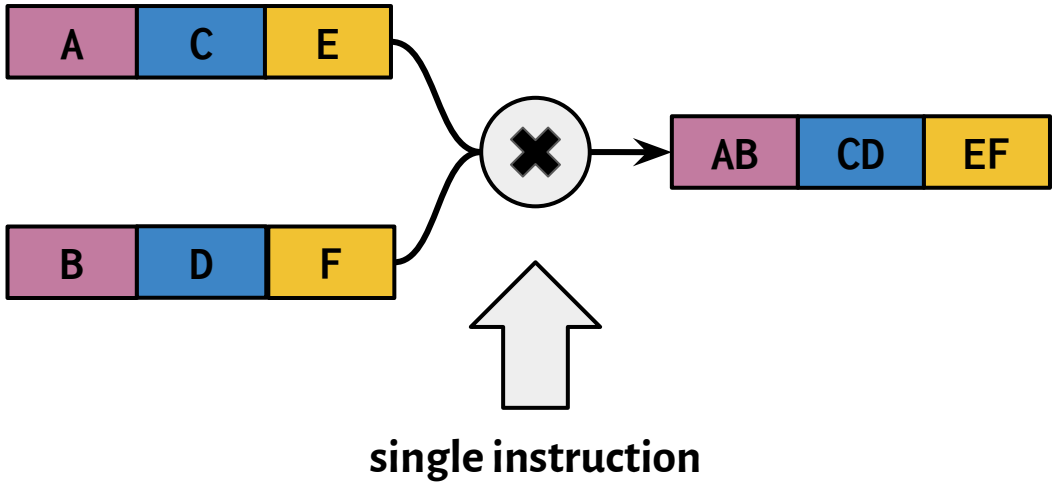
Parallelism



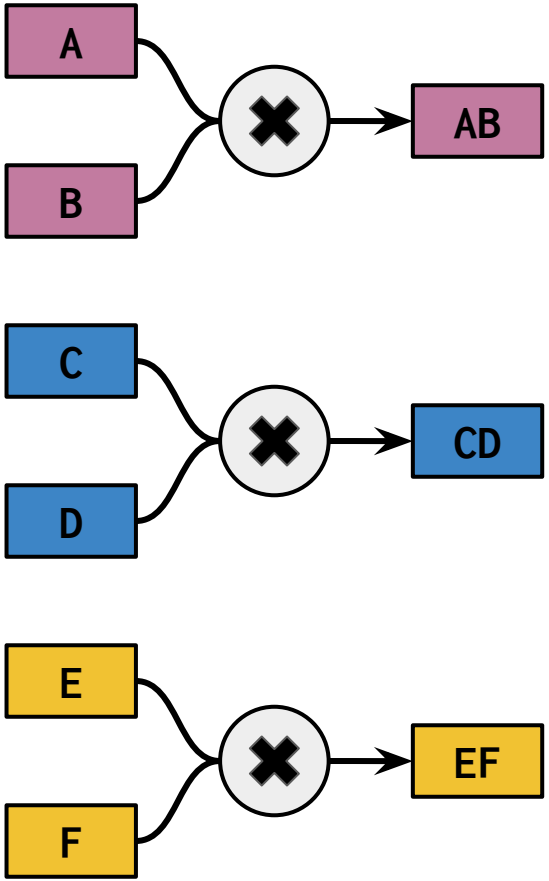


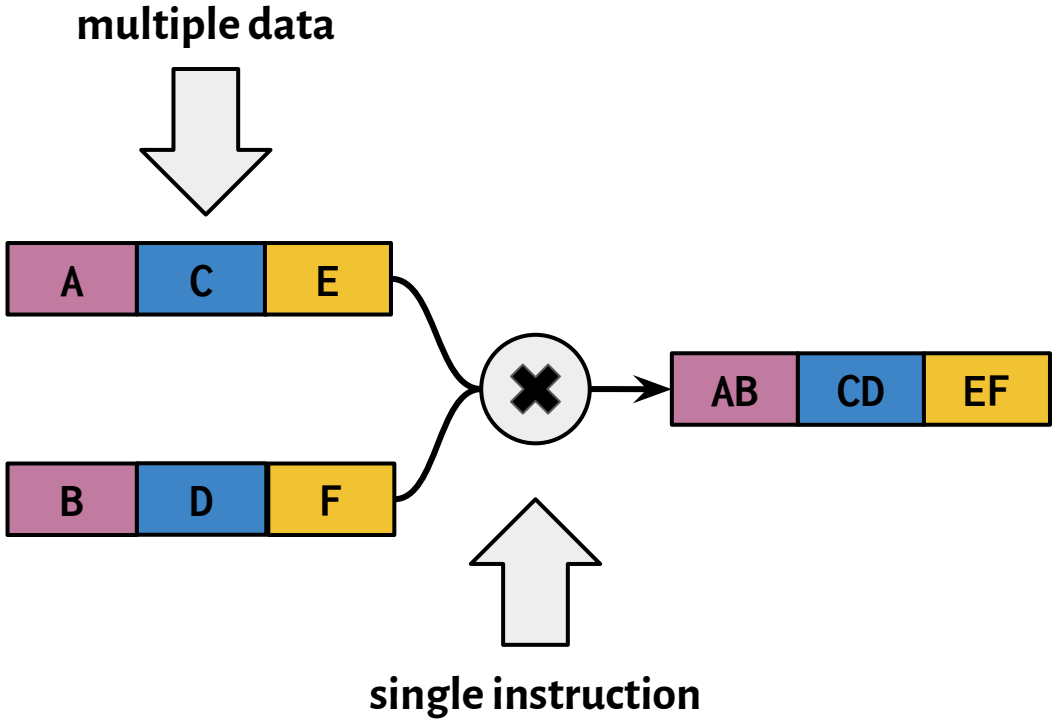
Parallelism





Parallelism





Optimize memory usage

Use stack instead of heap

Use fast-access memory

Optimize processor usage

Parallelize instructions

Improve the code structure

Optimize memory usage

Use stack instead of heap

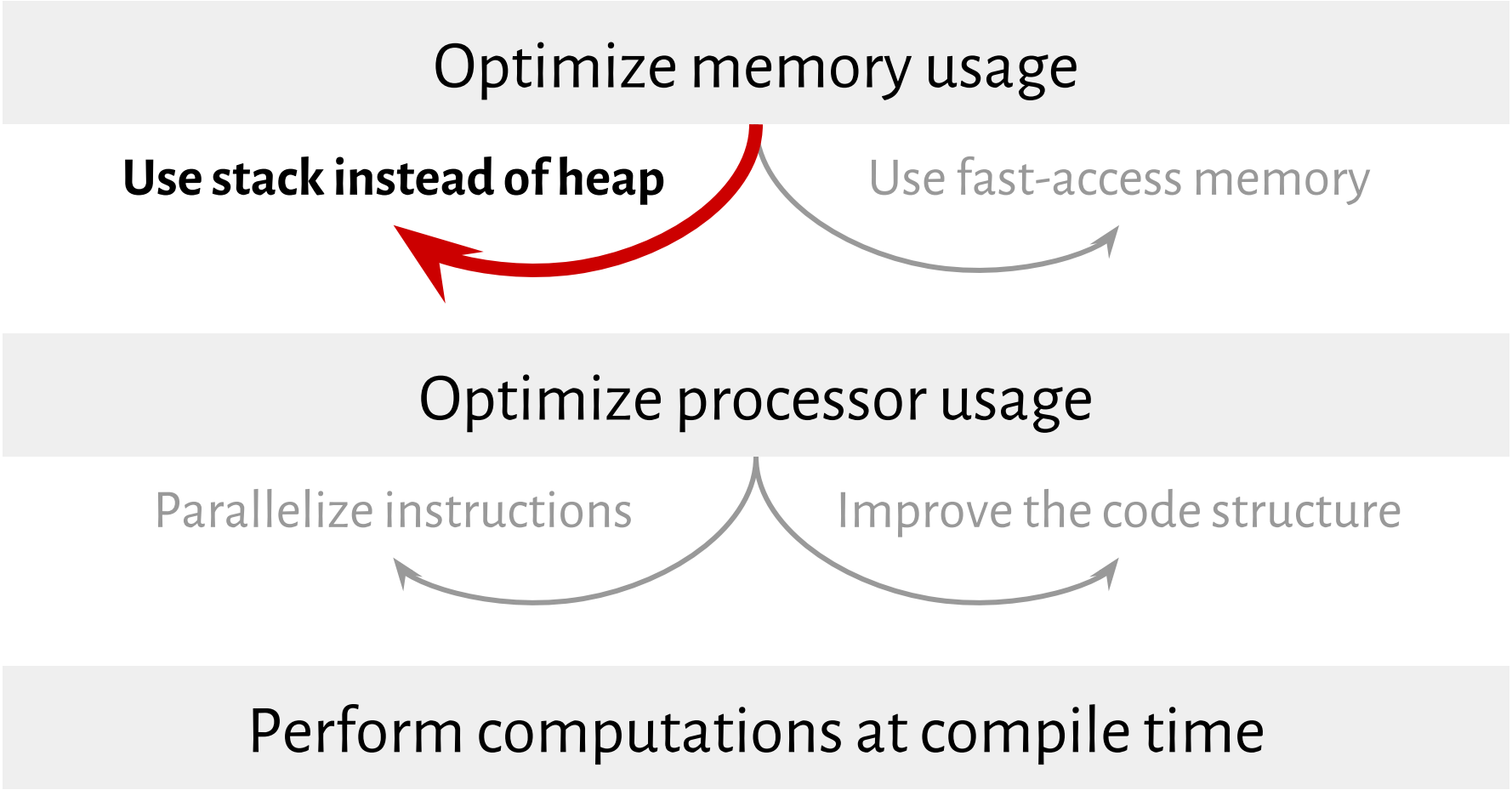
Use fast-access memory

Optimize processor usage

Parallelize instructions

Improve the code structure

Perform computations at compile time


Optimize memory usage
Use stack instead of heap
Use fast-access memory
Optimize processor usage
Parallelize instructions
Improve the code structure
Perform computations at compile time

What is allocated on the heap?

Any instances of dynamic size

Any instances of unspecified lifetime

Closure context

```
func bar() -> () -> Int {
    var x = 40
    modify(x: &x)
    let f = { return x + 2 }
    return f
}
```

Closure context

```
func bar() -> () -> Int {
    var x = 40
    modify(x: &x)
    let f = { return x + 2 }
    return f
}
```

Closure context

```
func bar() -> () -> Int {
    var x = 40
    modify(x: &x)
    let f = { return x + 2 }
    return f
}
```

Closure context

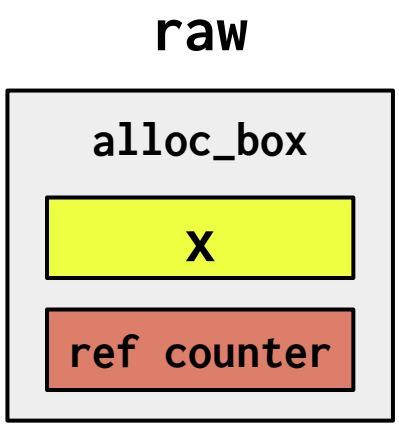
```
func bar() -> () -> Int {
    var x = 40
    modify(x: &x)
    let f = { return x + 2 }
    return f
}
```

Closure context

```
func bar() -> () -> Int {
    var x = 40
    modify(x: &x)
    let f = { return x + 2 }
    return f
}
```

Closure context

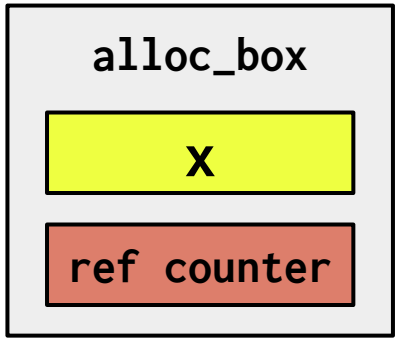
```
func bar() -> () -> Int {
    var x = 40
    modify(x: &x)
    let f = { return x + 2 }
    return f
}
```



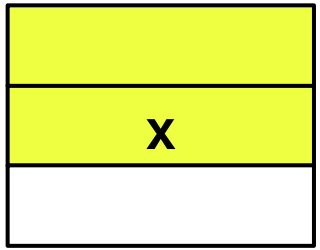
Capture promotion

```
func bar() -> () -> Int {
     var x = 40
     modify(x: &x)
     let f = { return x + 2 }
     return f
}
```

raw



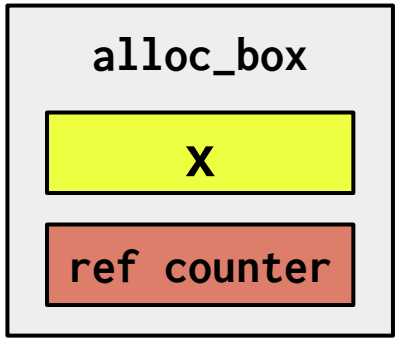
-Onone



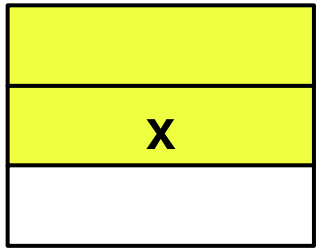
Capture promotion

```
func bar() -> () -> Int {
    var x = 40
    modify(x: &x)
    let f = { return x + 2 }
    return f
}
```

raw



-Onone



-O

How to spoil it?

```
func bar() -> () -> Int {
    var x = 40
    modify(x: &x)
    let f = { return x + 2 }
    return f
}
```

```
func bar() -> () -> Int {
    var x = 40
    let f = { return x + 2 }
    modify(x: &x)
    return f
}
```

How to spoil it?

```
func bar() -> () -> Int {
    var x = 40, y = 20, z = 10
    modify(x: &x, y: &y, z: &z)
    let f = { return x + y + z }
    return f
}
```

How to spoil it?

```
func modify(x: inout Int, y: inout Int, z: inout Int) {
    x += 2; y += 2; z += 2
}
```

```
func bar() -> () -> Int {
    var x = 40, y = 20, z = 10
    modify(x: &x, y: &y, z: &z)
    let f = { return x + y + z }
    return f
}
```

How to spoil it?

```
func modify(x: inout Int, y: inout Int, z: inout Int) {
    x += 2; y += 2; z += 2
}
```

```
func bar() -> () -> Int {
    var x = 40, y = 20, z = 10
    modify(x: &x, y: &y, z: &z)
    let f = { return x + y + z }
    return f
}
```

```
var sum = 0
for _ in 0...1_000_000:
    sum += f()
```

How to spoil it?

```
func modify(x: inout Int, y: inout Int, z: inout Int) {
    x += 2; y += 2; z += 2
}
```

```
func bar() -> () -> Int {
    var x = 40, y = 20, z = 10
    modify(x: &x, y: &y, z: &z)
    let f = { return x + y + z }
    return f
}
```

```
func bar() -> () -> Int {
    var x = 40, y = 20, z = 10
    let f = { return x + y + z }
    modify(x: &x, y: &y, z: &z)
    return f
}
```

```
var sum = 0
for _ in 0...1_000_000:
    sum += f()
```

How to spoil it?

```
func modify(x: inout Int, y: inout Int, z: inout Int) {
    x += 2; y += 2; z += 2
}
```

```
func bar() -> () -> Int {
    var x = 40, y = 20, z = 10
    modify(x: &x, y: &y, z: &z)
    let f = { return x + y + z }
    return f
}
```

```
func bar() -> () -> Int {
    var x = 40, y = 20, z = 10
    let f = { return x + y + z }
    modify(x: &x, y: &y, z: &z)
    return f
}
```

```
var sum = 0
for _ in 0...1_000_000:
    sum += f()
```

How to spoil it?

```
func modify(x: inout Int, y: inout Int, z: inout Int) {
    x += 2; y += 2; z += 2
}
```

```
func bar() -> () -> Int {
    var x = 40, y = 20, z = 10
    modify(x: &x, y: &y, z: &z)
    let f = { return x + y + z }
    return f
}
```

```
func bar() -> () -> Int {
    var x = 40, y = 20, z = 10
    let f = { return x + y + z }
    modify(x: &x, y: &y, z: &z)
    return f
}
```

```
var sum = 0
for _ in 0...1_000_000:
    sum += f()
```

-Onone: **3.45 s**

-Onone: **10.71 s**

How to spoil it?

```
func modify(x: inout Int, y: inout Int, z: inout Int) {
    x += 2; y += 2; z += 2
}
```

```
func bar() -> () -> Int {
    var x = 40, y = 20, z = 10
    modify(x: &x, y: &y, z: &z)
    let f = { return x + y + z }
    return f
}
```

```
func bar() -> () -> Int {
    var x = 40, y = 20, z = 10
    let f = { return x + y + z }
    modify(x: &x, y: &y, z: &z)
    return f
}
```

```
var sum = 0
for _ in 0...1_000_000:
    sum += f()
```

-Onone: **3.45 s** -O: **0.06 s**

-Onone: **10.71 s** -O: **9.90 s**

Suggestion

If possible, do not modify the variable after it is captured.

What is allocated on the heap?

Any instances of dynamic size

Any instances of unspecified lifetime

What is allocated on the heap?

Any instances of dynamic size

Any instances of unspecified lifetime

Classes

Stack promotion

```
class X {
     var m_x : Array<Int>
     init(_ n: Int, _ v: Int) {
          m_x = Array<Int>(repeating: n, count: v)
     }
}
```

Stack promotion

```
class X {
     var m_x : Array<Int>
     init(_ n: Int, _ v: Int) {
          m_x = Array<Int>(repeating: n, count: v)
     }
}
```

```
func createAndAccess() -> Int {
    let x = X(5, 10)
    return x.m_x[0]
}
```

Stack promotion

```
class X {
     var m_x : Array<Int>
     init(_ n: Int, _ v: Int) {
          m_x = Array<Int>(repeating: n, count: v)
     }
}
```

```
func createAndAccess() -> Int {
    let x = X(5, 10)
    return x.m_x[0]
}


func run() -> Int {
    return createAndAccess()
}
```

Stack promotion

```
class X {
     var m_x : Array<Int>
     init(_ n: Int, _ v: Int) {
          m_x = Array<Int>(repeating: n, count: v)
     }
}
```

```
func createAndAccess() -> Int {
     let x = X(5, 10)
     return x.m_x[0]
}


func run() -> Int {
     return createAndAccess()
}
```

```
var sum = 0
for _ in 1...1_000_000 {
     sum += run()
}
```

Stack promotion

```
class X {
     var m_x : Array<Int>
     init(_ n: Int, _ v: Int) {
          m_x = Array<Int>(repeating: n, count: v)
     }
}
```

```
func createAndAccess() -> Int {
     let x = X(5, 10)
     return x.m_x[0]
}


func run() -> Int {
     return createAndAccess()
}
```

```
var sum = 0
for _ in 1...1_000_000 {
     sum += run()
}
```

-O and without optimization: **163 ms**

Stack promotion

```
class X {
     var m_x : Array<Int>
     init(_ n: Int, _ v: Int) {
          m_x = Array<Int>(repeating: n, count: v)
     }
}
```

```
func createAndAccess() -> Int {
     let x = X(5, 10)
     return x.m_x[0]
}


func run() -> Int {
     return createAndAccess()
}
```

```
var sum = 0
for _ in 1...1_000_000 {
     sum += run()
}
```

-O and without optimization: **163 ms**

-O and with optimization: **92 ms**

Stack promotion

```
class X {
     var m_x : Array<Int>
     init(_ n: Int, _ v: Int) {
          m_x = Array<Int>(repeating: n, count: v)
     }
}
```

```
func createAndAccess() -> Int {
    let x = X(5, 10)
    return x.m_x[0]
}


func run() -> Int {
    return createAndAccess()
}
```

```
func createAndReturn() -> X {
    let x = X(5, 10)
    return x
}


func run() -> Int {
     return createAndReturn().m_x[0]
}
```

Stack promotion

```
class X {
     var m_x : Array<Int>
     init(_ n: Int, _ v: Int) {
          m_x = Array<Int>(repeating: n, count: v)
     }
}
```

```
func createAndAccess() -> Int {
    let x = X(5, 10)
    return x.m_x[0]
}

func run() -> Int {
    return createAndAccess()
}
```

```
func createAndAccess() -> Int {
    let x = X(5, 10)
    return access(x)
}

func access(_ x: X) -> Int {
     return x.m_x[0]
}
```

Stack promotion

```
class X {
     let m_x : Array<Int>
     init(_ n: Int, _ v: Int) {
          m_x = Array<Int>(repeating: n, count: v)
     }
}
```

```
func createAndAccess() -> Int {
    let x = X(5, 10)
    return x.m_x[0]
}

func run() -> Int {
    return createAndAccess()
}
```

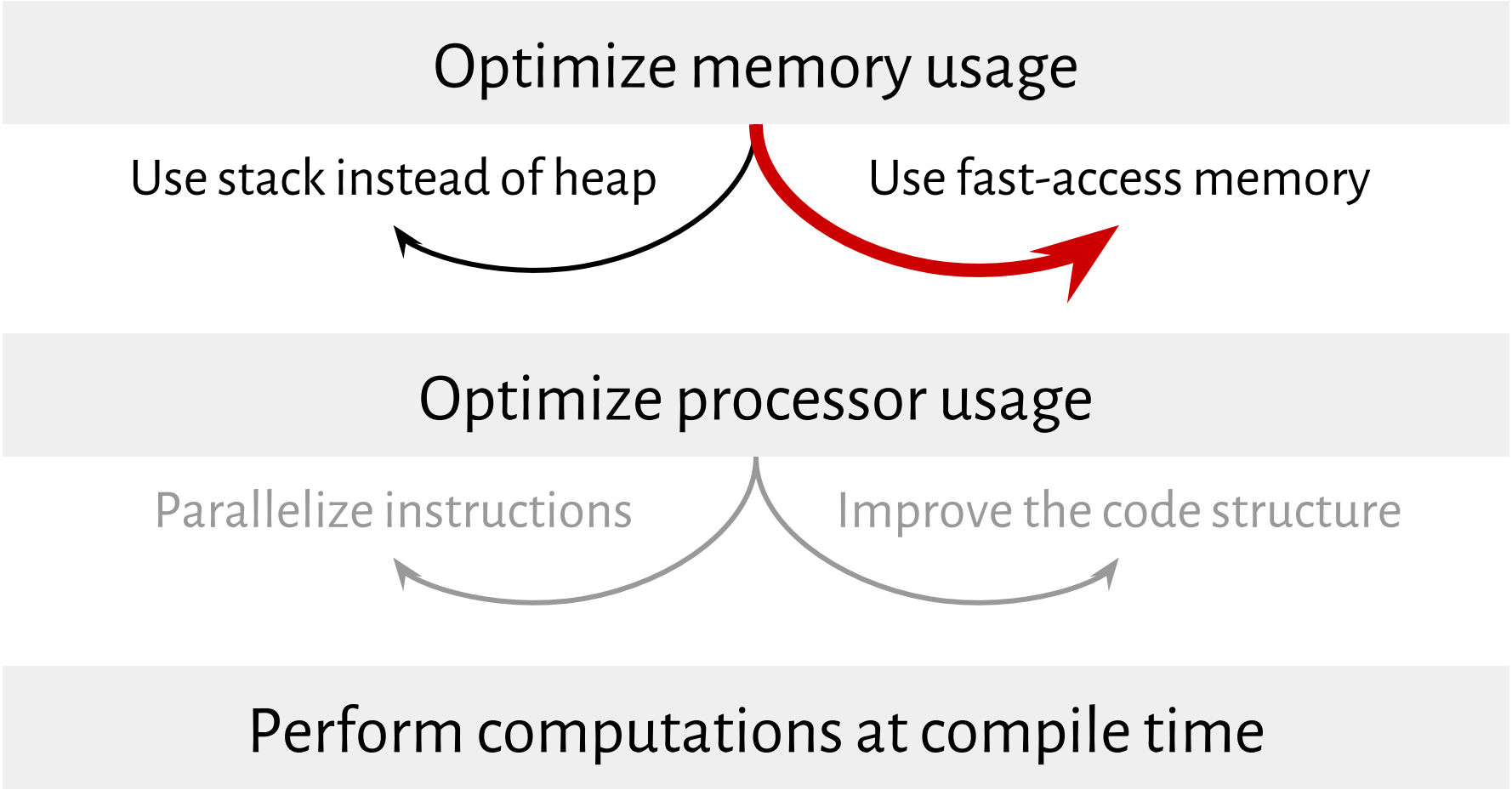
```
func createAndAccess() -> Int {
    let x = X(5, 10)
    return access(x)
}

func access(_ x: X) -> Int {
     return x.m_x[0]
}
```

Suggestion

Try to limit an object lifetime to a single function.

If possible, **return values** rather than objects.

Make sure that all read-only properties are declared with **`let`**.

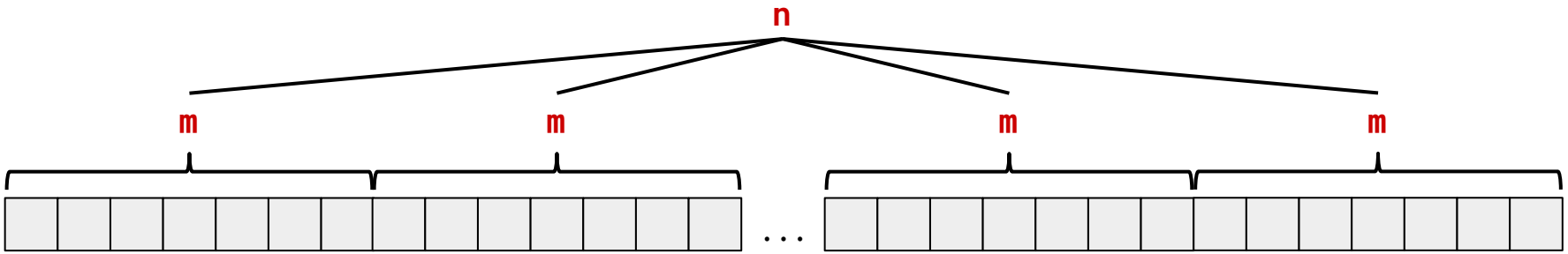

Optimize memory usage
Use stack instead of heap
Use fast-access memory
Optimize processor usage
Parallelize instructions
Improve the code structure
Perform computations at compile time

Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```

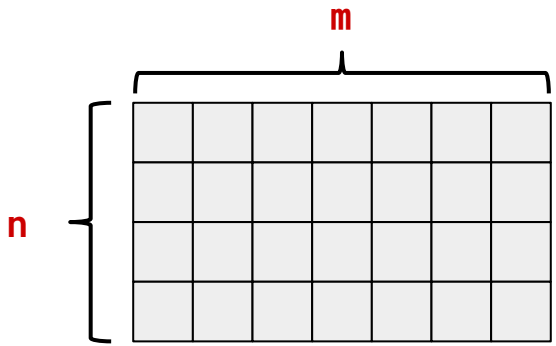
Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```



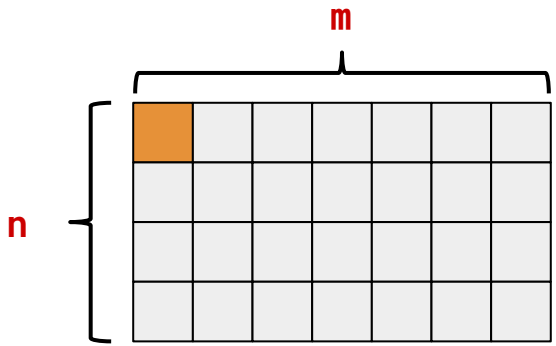
Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```



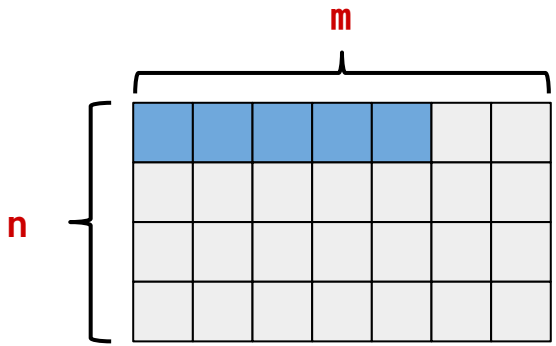
Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```



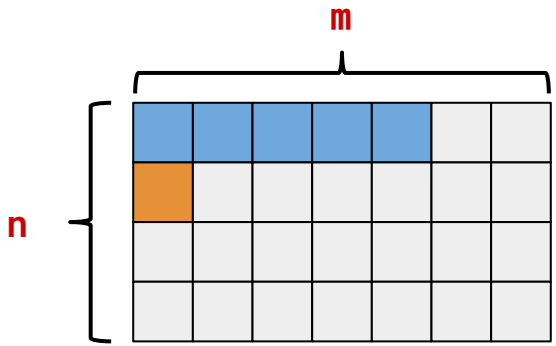
Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```



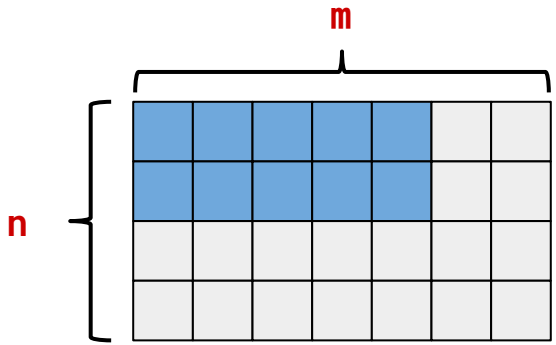
Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```



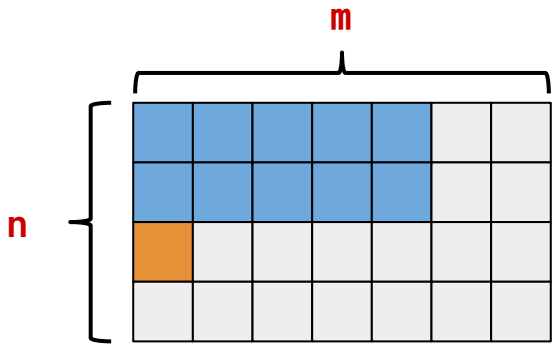
Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```



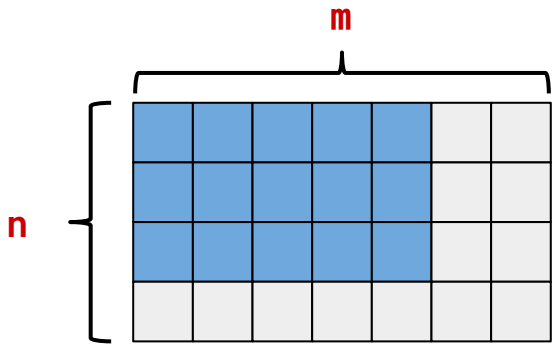
Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```



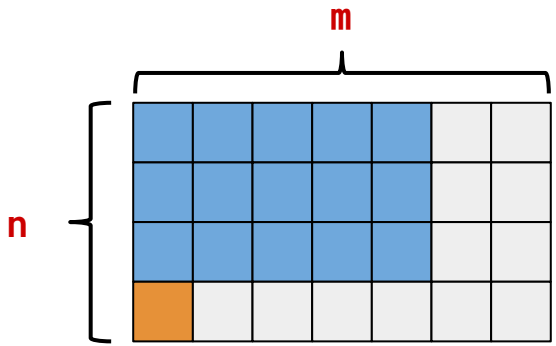
Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```



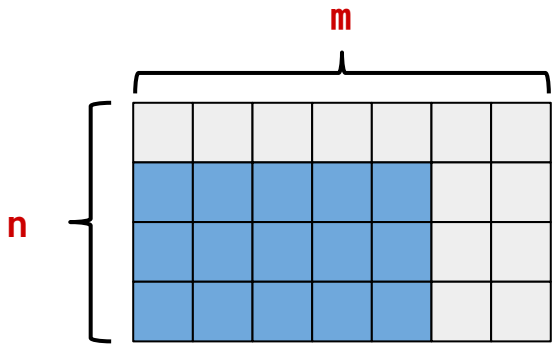
Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```



Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```



Spatial locality

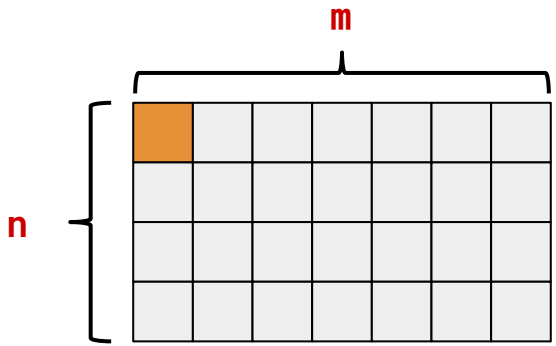
```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```

```
for i in 0..<n {
     for j in 0..<m {
          tabA[i][j] = i * j
     }
}
```

Spatial locality

```
for j in 0..<m {
    for i in 0..<n {
        tabA[i][j] = i * j
    }
}
```

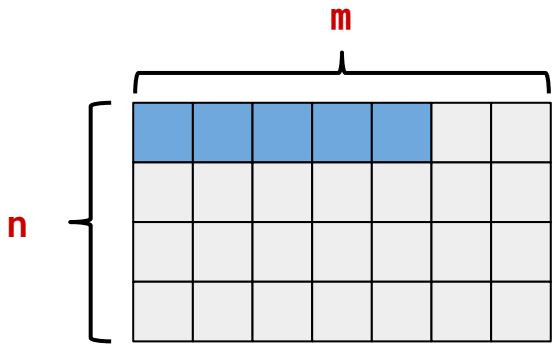
```
for i in 0..<n {
    for j in 0..<m {
        tabA[i][j] = i * j
    }
}
```



Spatial locality

```
for j in 0..<m {
    for i in 0..<n {
        tabA[i][j] = i * j
    }
}
```

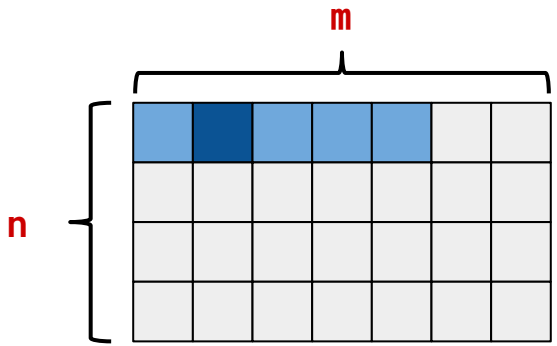
```
for i in 0..<n {
    for j in 0..<m {
        tabA[i][j] = i * j
    }
}
```



Spatial locality

```
for j in 0..<m {
    for i in 0..<n {
        tabA[i][j] = i * j
    }
}
```

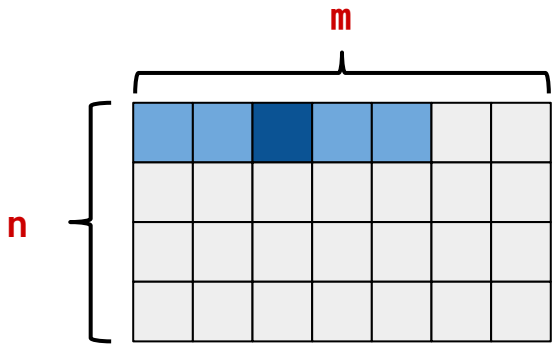
```
for i in 0..<n {
    for j in 0..<m {
        tabA[i][j] = i * j
    }
}
```



Spatial locality

```
for j in 0..<m {
    for i in 0..<n {
        tabA[i][j] = i * j
    }
}
```

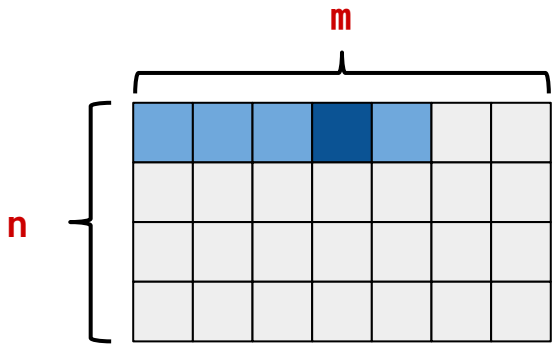
```
for i in 0..<n {
    for j in 0..<m {
        tabA[i][j] = i * j
    }
}
```



Spatial locality

```
for j in 0..<m {
    for i in 0..<n {
        tabA[i][j] = i * j
    }
}
```

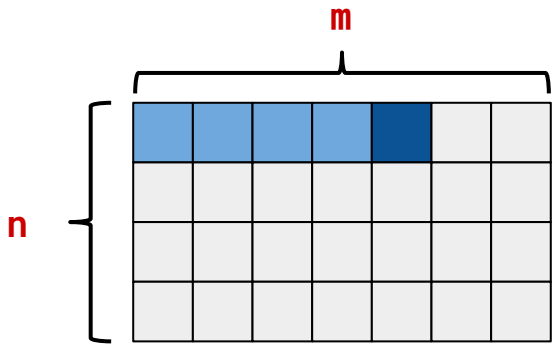
```
for i in 0..<n {
    for j in 0..<m {
        tabA[i][j] = i * j
    }
}
```



Spatial locality

```
for j in 0..<m {
    for i in 0..<n {
        tabA[i][j] = i * j
    }
}
```

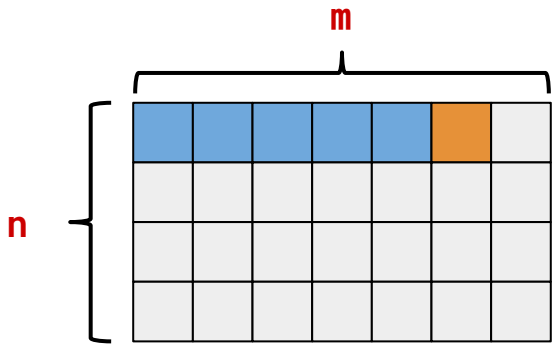
```
for i in 0..<n {
    for j in 0..<m {
        tabA[i][j] = i * j
    }
}
```



Spatial locality

```
for j in 0..<m {
    for i in 0..<n {
        tabA[i][j] = i * j
    }
}
```

```
for i in 0..<n {
    for j in 0..<m {
        tabA[i][j] = i * j
    }
}
```



Spatial locality

```
for j in 0..<m {
    for i in 0..<n {
        tabA[i][j] = i * j
    }
}
```

```
for i in 0..<n {
    for j in 0..<m {
        tabA[i][j] = i * j
    }
}
```

n = 3000

m = 5000

560 ms

Spatial locality

```
for j in 0..<m {
    for i in 0..<n {
        tabA[i][j] = i * j
    }
}
```

```
for i in 0..<n {
    for j in 0..<m {
        tabA[i][j] = i * j
    }
}
```

n = 3000

m = 5000

560 ms

70 ms

Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```

```
for i in 0..<n {
     for j in 0..<m {
          tabA[i][j] = i * j
     }
}
```

```
swiftc -O -Xllvm -enable-loopinterchange file.swift
```

Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```

```
for i in 0..<n {
     for j in 0..<m {
          tabA[i][j] = i * j
     }
}
```

```
swiftc -Ounchecked -Xllvm -enable-loopinterchange file.swift
```

Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```

```
for i in 0..<n {
     for j in 0..<m {
          tabA[i][j] = i * j
     }
}
```

```
swiftc -Ounchecked -Xllvm -enable-loopinterchange file.swift
```

```
llvm/lib/Transforms/Scalar/LoopInterchange.cpp
```

Spatial locality

```
for j in 0..<m {
     for i in 0..<n {
          tabA[i][j] = i * j
     }
}
```

```
for i in 0..<n {
     for j in 0..<m {
          tabA[i][j] = i * j
     }
}
```

```
swiftc -Ounchecked -Xllvm -enable-loopinterchange file.swift
```

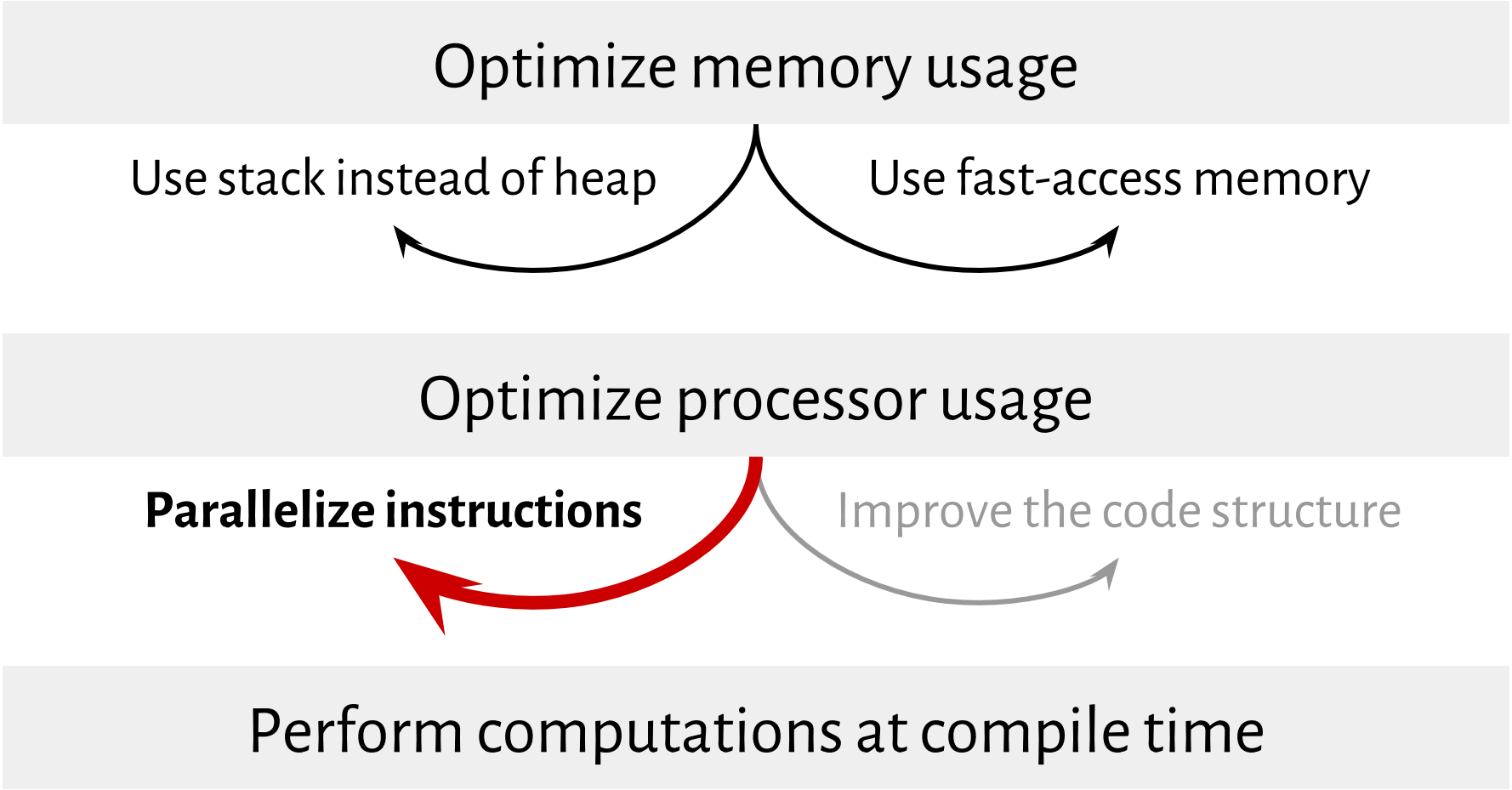
```
llvm/lib/Transforms/Scalar/LoopInterchange.cpp
```

```
// TODO: Handle flow dependence.
```

Suggestion

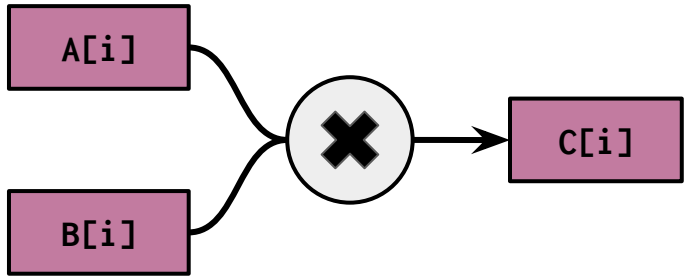
Monitor **both** compilers for any updates.

As for now: take care of **data locality** by yourself.


Optimize memory usage
Use stack instead of heap
Use fast-access memory
Optimize processor usage
Parallelize instructions
Improve the code structure
Perform computations at compile time

Loop vectorization

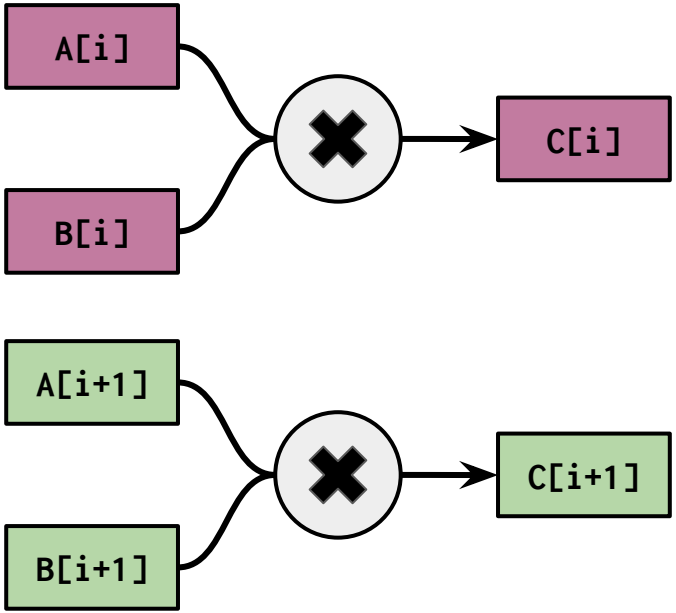
```
for i in 0..<10_000_000 {
```




```
}
```

Loop unrolling

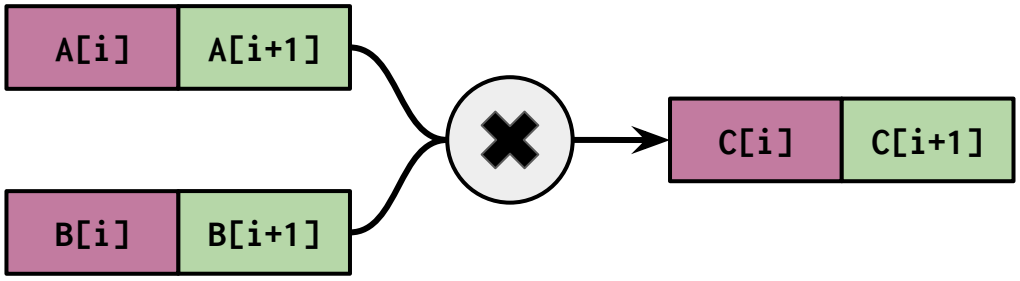
```
for i in 0..<5_000_000 {
```




```
}
```

Vectorization

```
for i in 0..<5_000_000 {
```




```
}
```

Any improvements?

```
for i in 0..<10_000_000 {
    tabC[i] = 2.0 * tabA[i]
    tabC[i] /= (1.0 + tabB[i])
}
```

-Ounchecked and without optimization: **130 ms**

Any improvements?

```
for i in 0..<10_000_000 {
    tabC[i] = 2.0 * tabA[i]
    tabC[i] /= (1.0 + tabB[i])
}
```

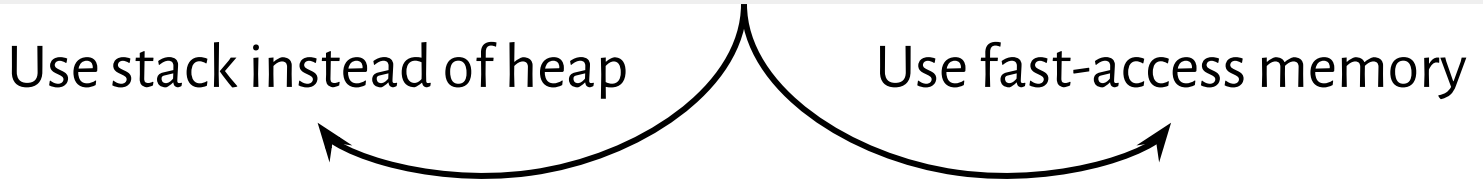
-Ounchecked and without optimization: **130 ms**

-Ounchecked and with optimization: **55 ms**

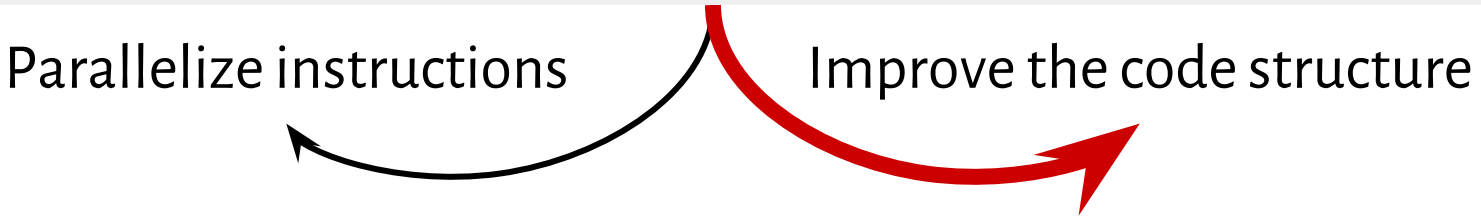
Suggestion

Use -Ounchecked for critical sections of code.

Optimize memory usage



Optimize processor usage



Perform computations at compile time

Performance inlining

```
func getDiff(a: Int, b: Int) -> Int {
    if (a > b) {
        return a - b
    }
    return b - a
}

var a = 7
var b = 3
var c = getDiff(a: a, b: b)
```

Performance inlining

```
func getDiff(a: Int, b: Int) -> Int {
    if (a > b) {
        return a - b
    }
    return b - a
}

var a = 7
var b = 3
var c = getDiff(a: a, b: b)
```

Performance inlining

```
func getDiff(a: Int, b: Int) -> Int {
    if (a > b) {
        return a - b
    }
    return b - a
}

var a = 7
var b = 3
var c = getDiff(a: a, b: b)
```

Performance inlining

```
func getDiff(a: Int, b: Int) -> Int {
    if (a > b) {
        return a - b
    }
    return b - a
}
```

```
var a = 7
var b = 3
var c = getDiff(a: a, b: b)
```

Performance inlining

```
func getDiff(a: Int, b: Int) -> Int {
    if (a > b) {
        return a - b
    }
    return b - a
}

var a = 7
var b = 3
var c = getDiff(a: a, b: b)
```

```
var a = 7
var b = 3
var c: Int
if (a > b) {
    c = a - b
} else {
    c = b - a
}
```

swiftc -O file.swift

Performance inlining

```
func getDiff(a: Int, b: Int) -> Int {
    if (a > b) {
        return a - b
    }
    return b - a
}

var a = 7
var b = 3
var c = getDiff(a: a, b: b)
```

```
var a = 7
var b = 3
var c: Int
if (a > b) {
    c = a - b
} else {
    c = b - a
}
```

swiftc -O file.swift

Performance inlining

```
func getDiff(a: Int, b: Int) -> Int {
    if (a > b) {
        return a - b
    }
    return b - a
}

var a = 7
var b = 3
var c = getDiff(a: a, b: b)
```

```
var a = 7
var b = 3
var c: Int
if (a > b) {
    c = a - b
} else {
    c = b - a
}
```

swiftc -O file.swift

Performance inlining

```
func getDiff(a: Int, b: Int) -> Int {
    if (a > b) {
        return a - b
    }
    return b - a
}

var a = 7
var b = 3
var c = getDiff(a: a, b: b)
```

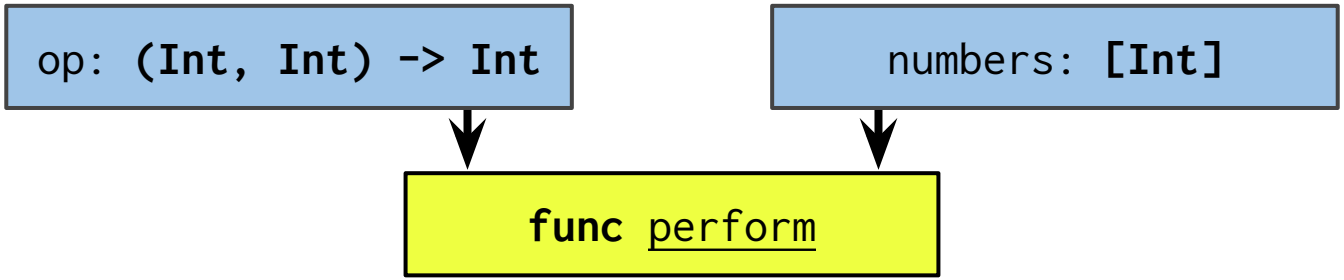
```
var a = 7
var b = 3
var c = 4
```

swiftc -O file.swift

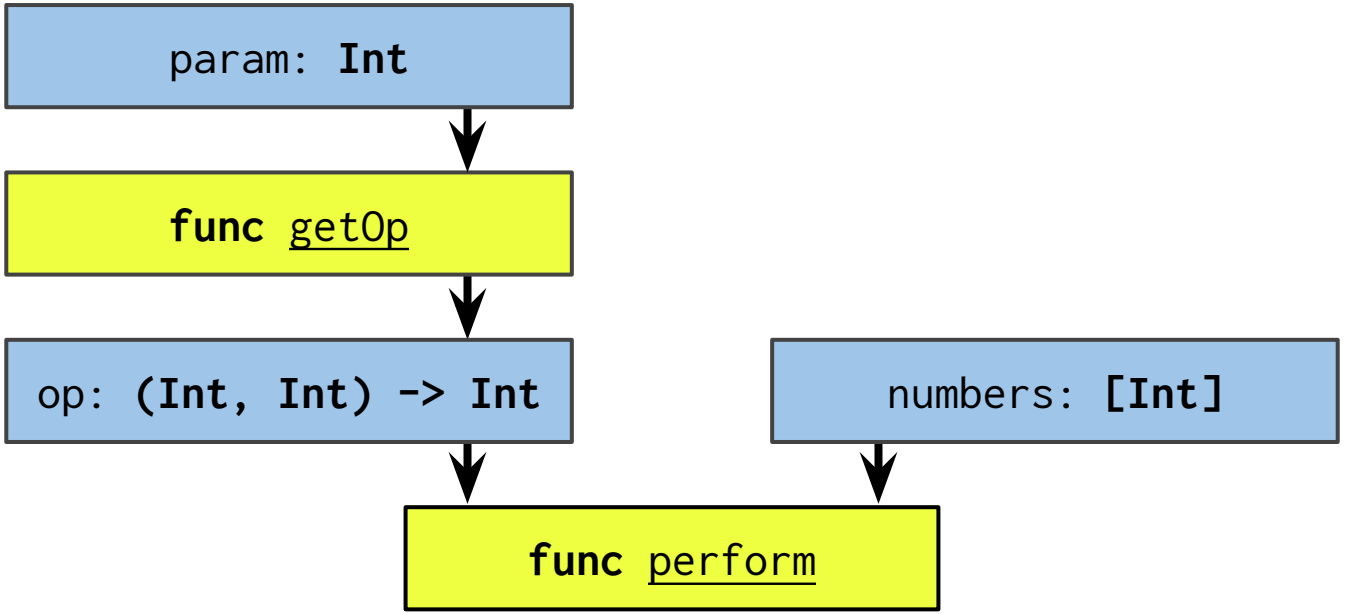
Performance inlining

What about closures?

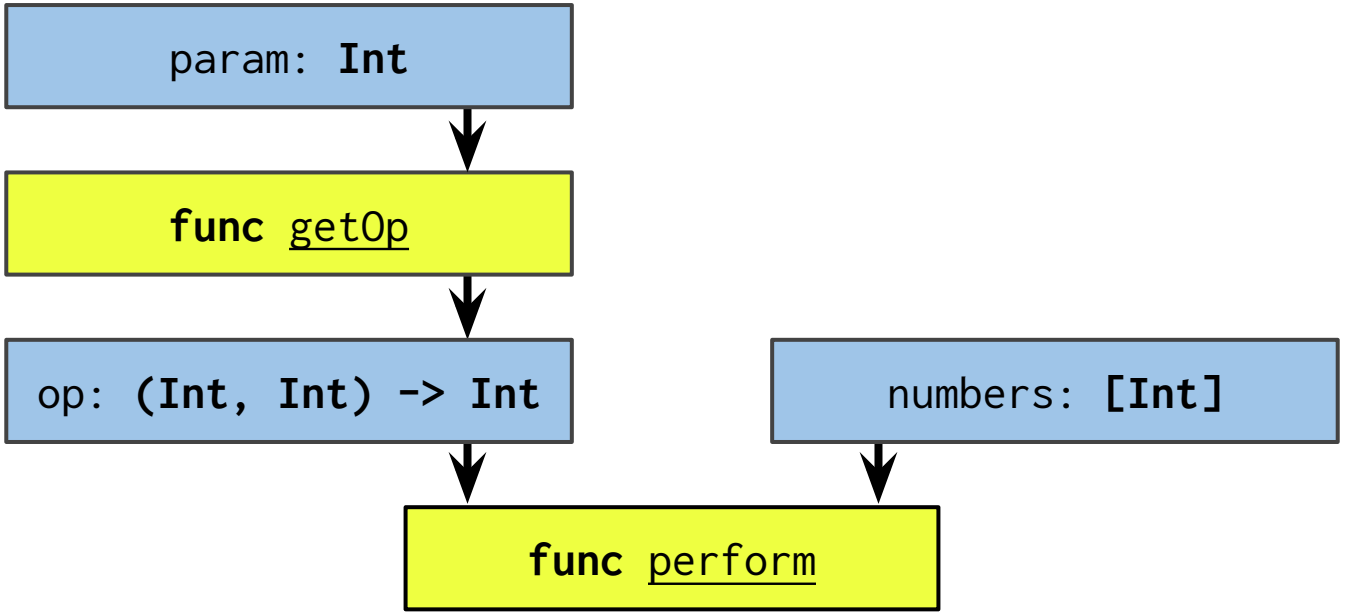
Closure specialization



Closure specialization

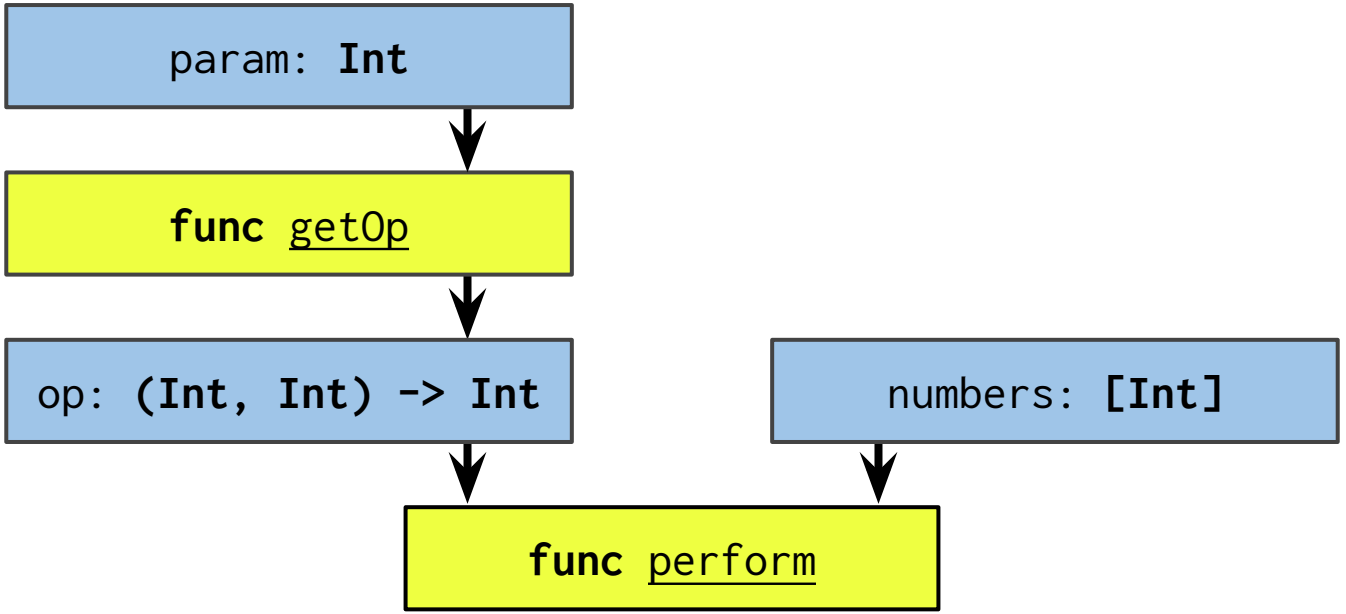


Closure specialization




```
perform(op: getOp(param: n), on: numbers)
```

Closure specialization




```
perform(op: getOp(param: n), on: numbers)
```

specialized func performGetOp

Closure specialization

```
func perform(op: (Int, Int) -> Int, on numbers: [Int]) -> Int {
    var accumulator = 1
    for n in numbers {
      accumulator = op(accumulator, n)
    }
    return accumulator
}
```

```
func getOp(_ c: Int) -> (Int, Int) -> Int {
     let d = 2 * c
     return {(a: Int, b: Int) -> Int in return a + b + d }
}
```

```
perform(op: getOp(5), on: numbers)
```

Any improvements?

-O and without optimization: **45 ms**

Any improvements?

-O and without optimization:	**45 ms**
-O and with optimization:	**17 ms**

Any improvements?

-O and without optimization: **45 ms**

-O and with optimization: **17 ms**

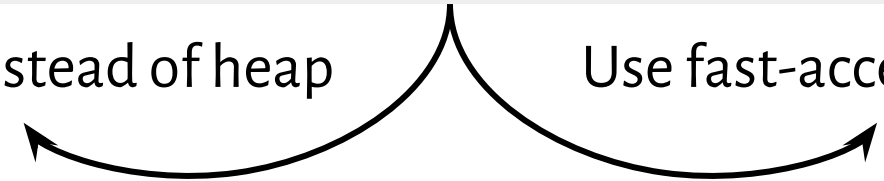
Again, you can easily spoil it:

```
func getOp(_ c: Int) -> (Int, Int) -> Int {
    var d = 2 * c
    let result = {(a: Int, b: Int) -> Int in return a + b + d }
    if (d != 2 * c) {
        d = 2 * c
    }
}
```

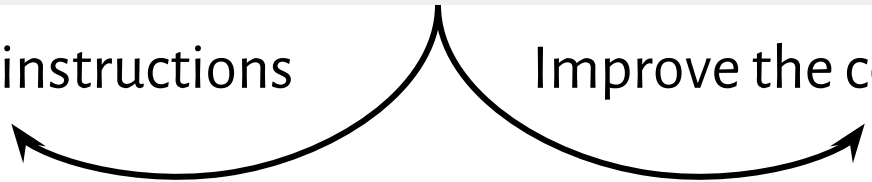
Suggestion

If possible, do not modify the variable after it is captured.

Optimize memory usage



Optimize processor usage



Perform computations at compile time

Dynamic dispatch

```
class CarModel {
     var position: Int = 0
     var velocity: Int = 0

     func move() { … }
}
```

Dynamic dispatch

```
class CarModel {
     var position: Int = 0
     var velocity: Int = 0

     func move() { … }
}
```

```
class TurboCarModel : CarModel {
     var turboMode: Bool = false

     override func move() { … }
}
```

Dynamic dispatch

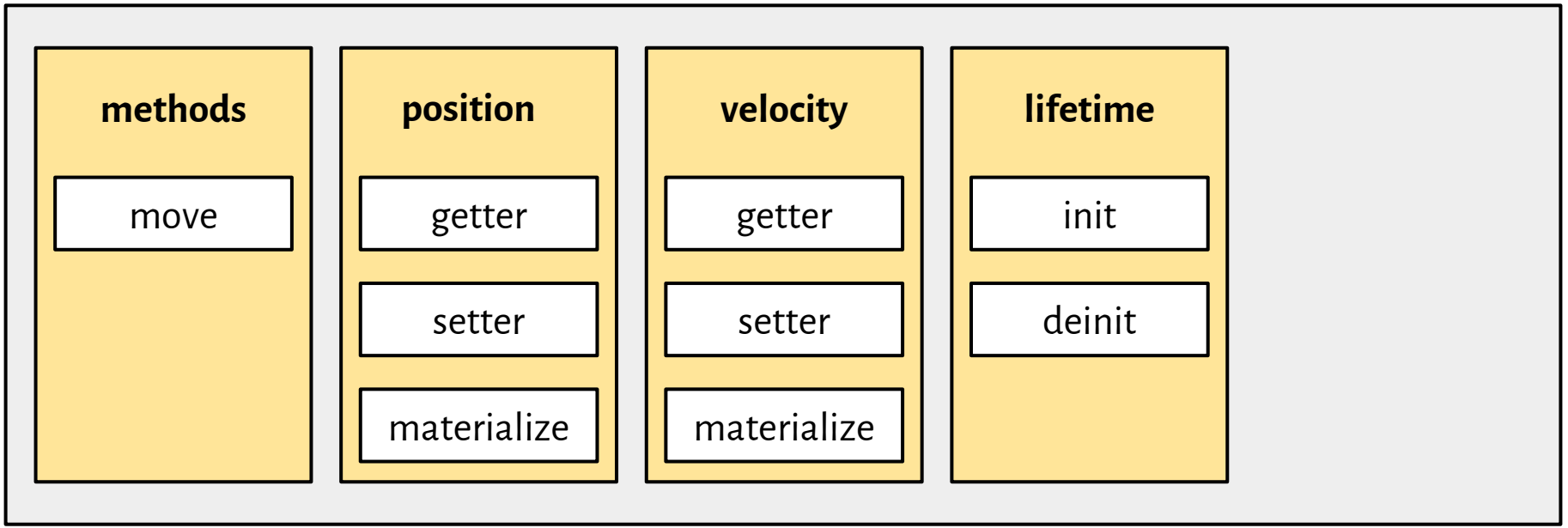
```
class CarModel {
     var position: Int = 0
     var velocity: Int = 0

     func move() { … }
}
```

```
class TurboCarModel : CarModel {
     var turboMode: Bool = false

     override func move() { … }
}
```

```
func drive(c: CarModel) {
     c.move()
}
```

VTable

VTable


methods
move
position
getter
setter
materialize
velocity
getter
setter
materialize
lifetime
init
deinit

VTable

methods	position	velocity	lifetime	turboMode
move	getter	getter	init	getter
	setter	setter	deinit	setter
	materialize	materialize		materialize

Dynamic dispatch

```
override func move() {
    if (turboMode) {
        position += 2 * velocity
    } else {
        super.move()
    }
}
```

Dynamic dispatch

```
override func move() {
    if (turboMode) {
        position += 2 * velocity
    } else {
        super.move()
    }
}
```

Dynamic dispatch

```
override func move() {
    if (turboMode) {
        position += 2 * velocity
    } else {
        super.move()
    }
}
```

Dynamic dispatch

```
override func move() {
    if (turboMode) {
        position += 2 * velocity
    } else {
        super.move()
    }
}
```

Dynamic dispatch

```
override func move() {
    if (turboMode) {
        position += 2 * velocity
    } else {
        super.move()
    }
}
```

go to the address

go to the vtable (dynamic dispatch)

Limit the dynamic dispatch

```
class CarModel {
     final var position: Int = 0
     final var velocity: Int = 0

     func move() { … }
}
```

```
class TurboCarModel : CarModel {
     final var turboMode: Bool = false

     override func move() { … }
}
```

Protocols

```
protocol Movable { func move() }
```

Protocols

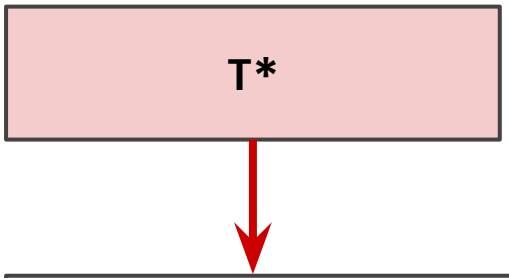
```
protocol Movable { func move() }
```

```
class CarModel: Movable{
     var position: Int = 0
     var velocity: Int = 0

     func move() { … }
}
```

Protocols

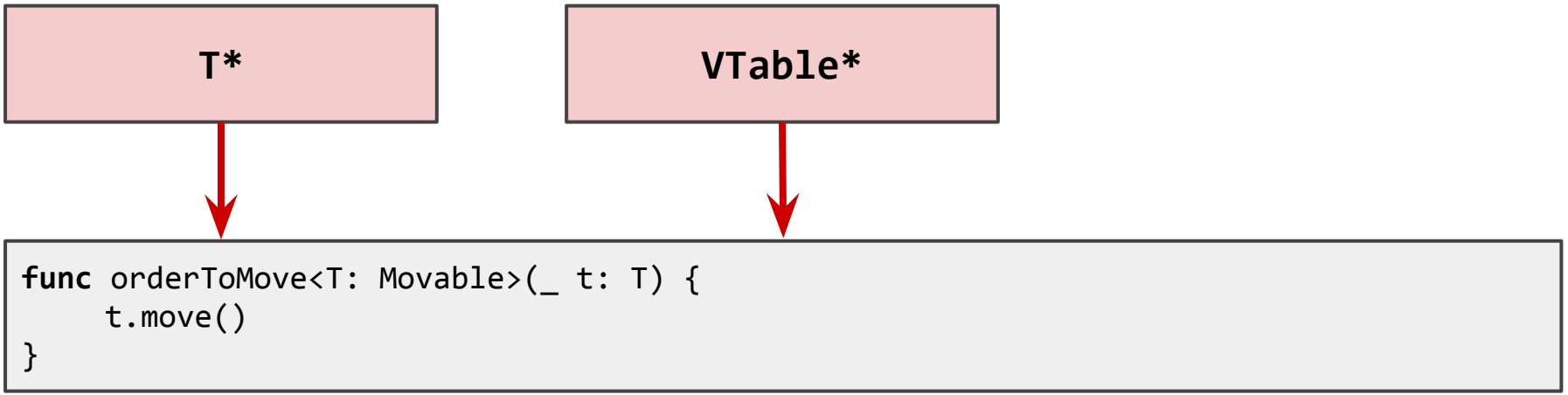
```
func orderToMove<T: Movable>(_ t: T) {
    t.move()
}
```

Protocols



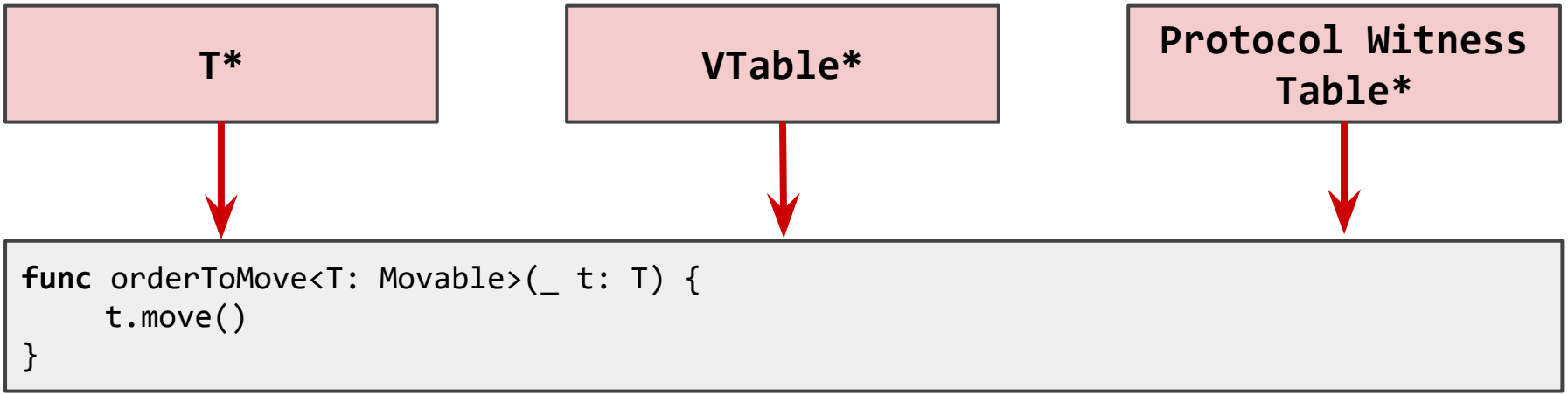

```
func orderToMove<T: Movable>(_ t: T) {
     t.move()
}
```

Protocols

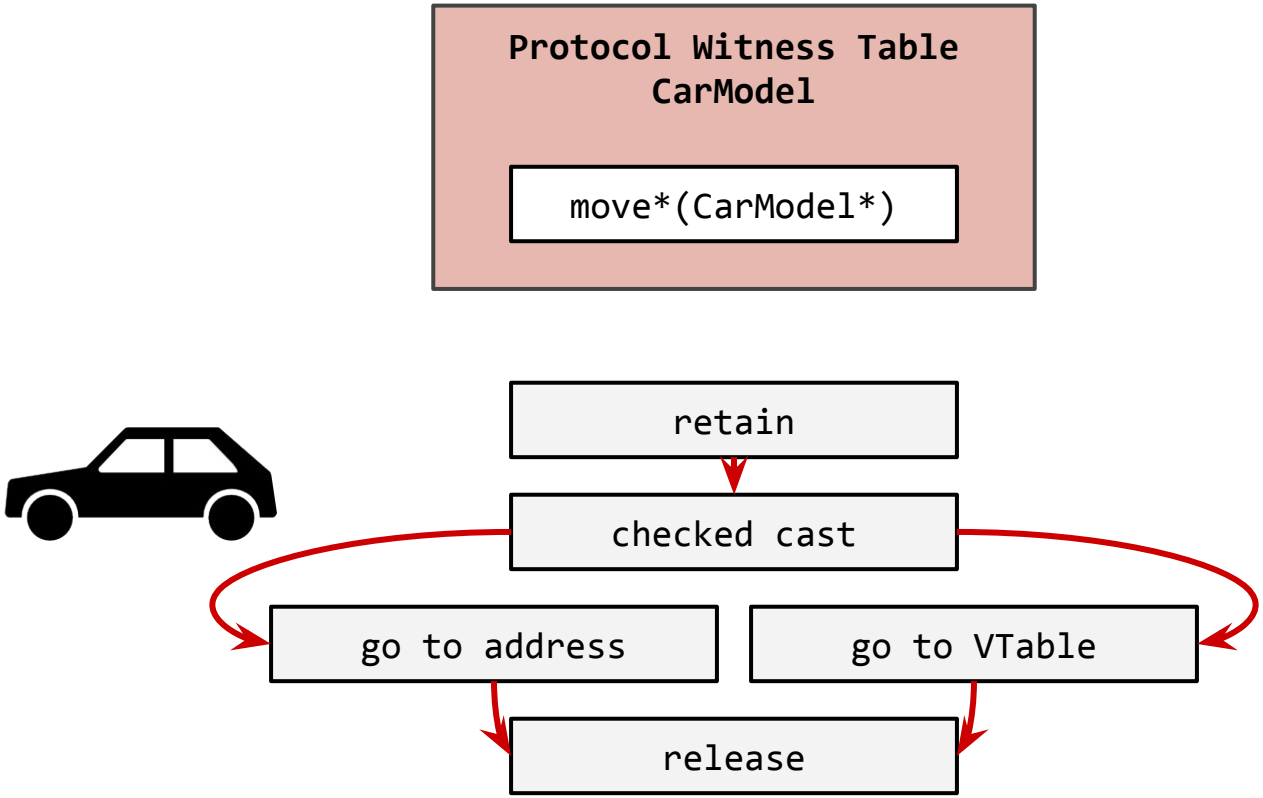



```
func orderToMove<T: Movable>(_ t: T) {
    t.move()
}
```

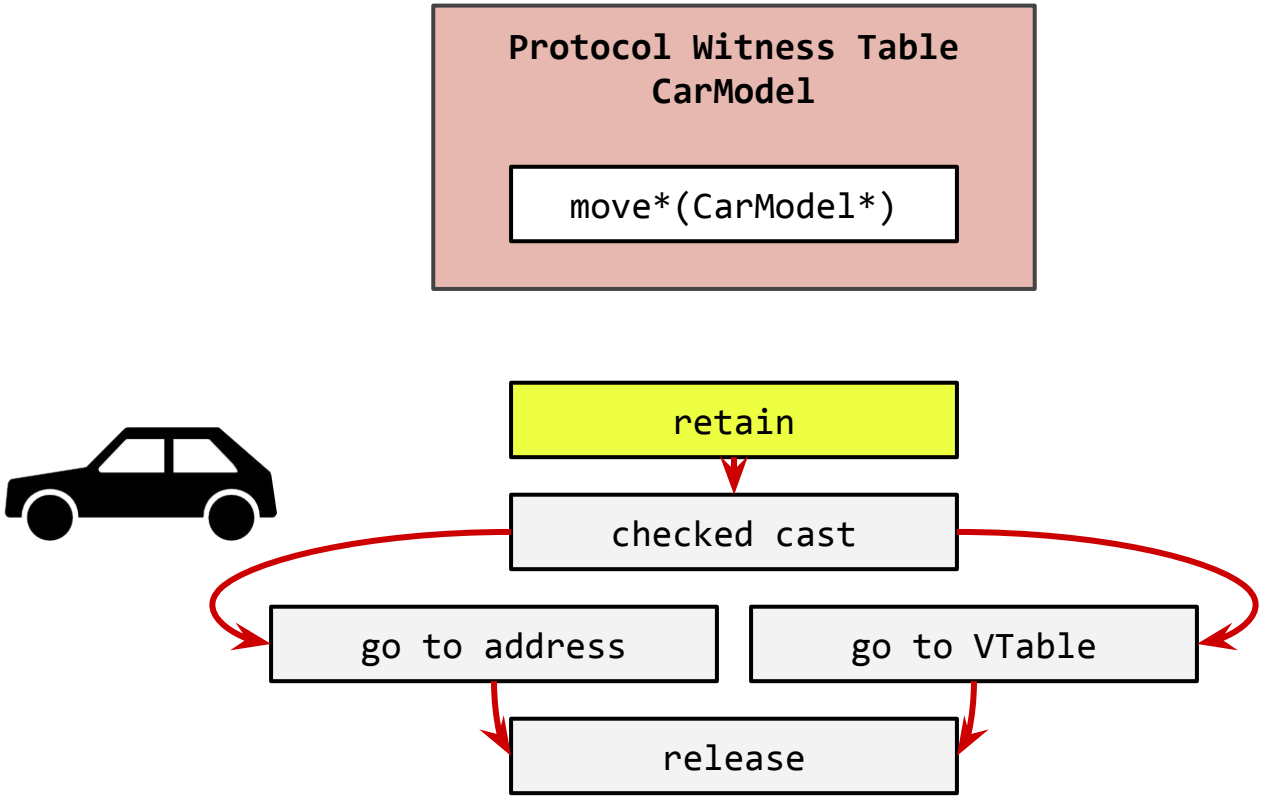
Protocols


T*
VTable*
Protocol Witness Table*
func orderToMove<T: Movable>(_ t: T) {
t.move()
}

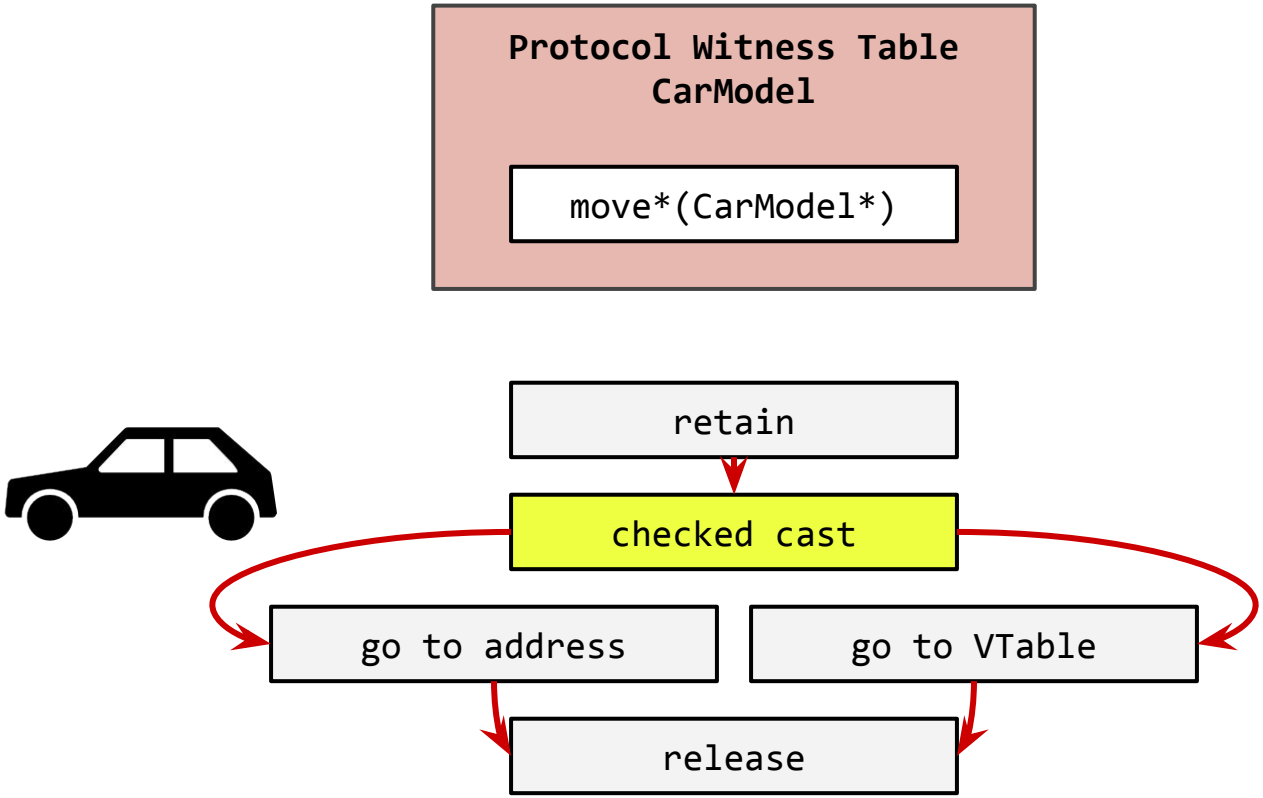
Protocol Witness Table



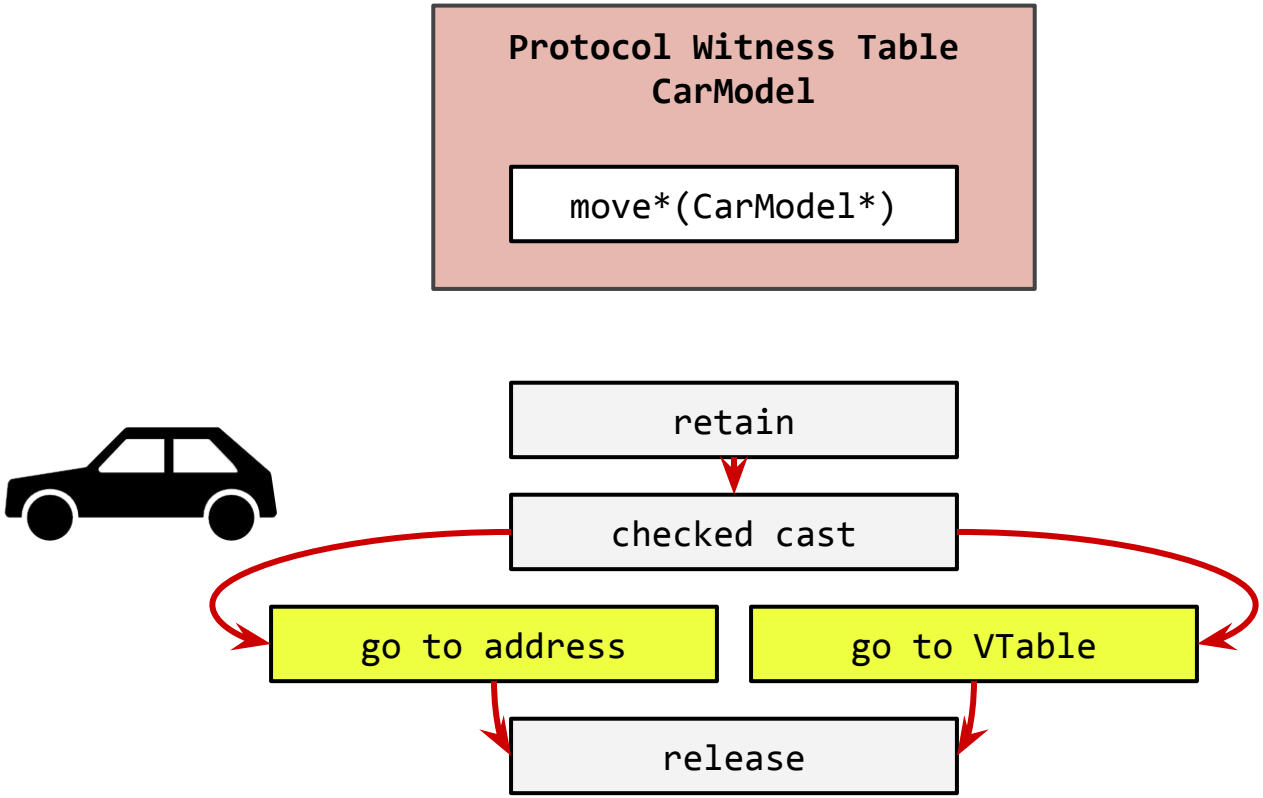
Protocol Witness Table



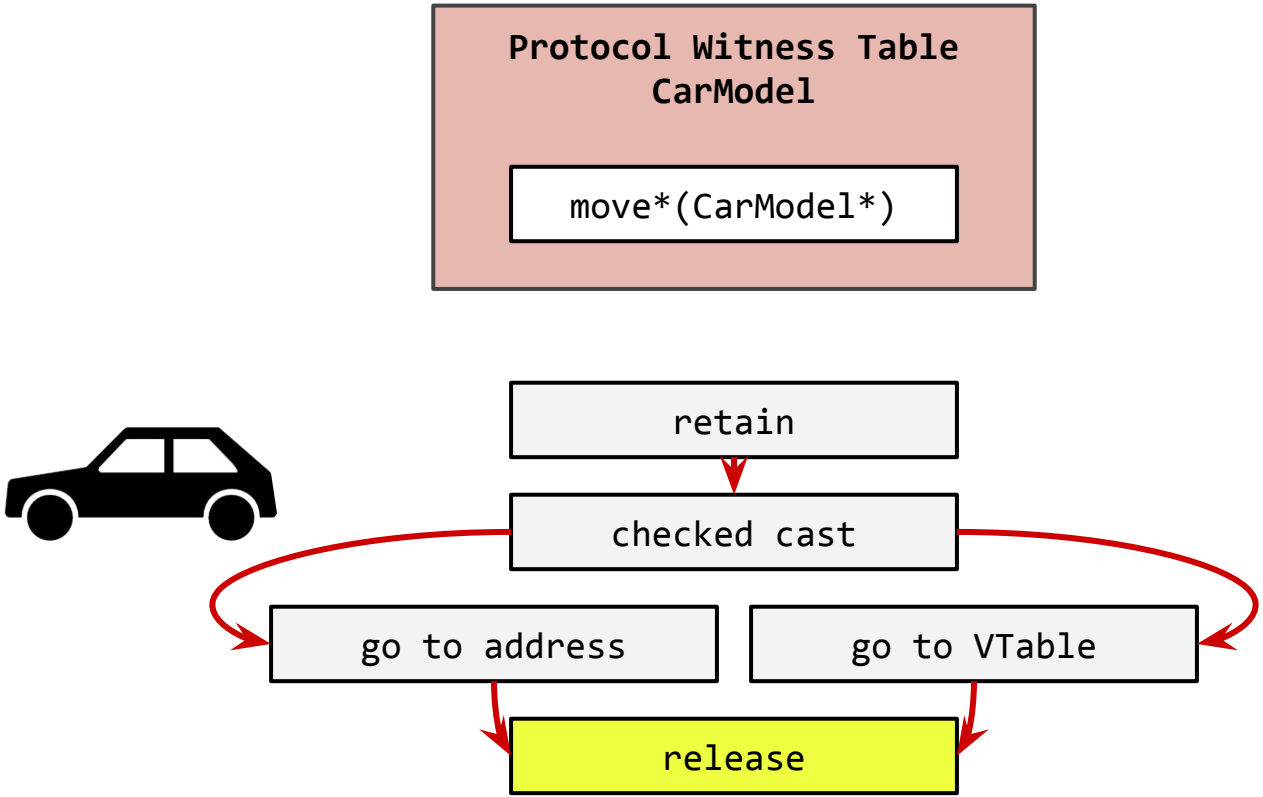
Protocol Witness Table



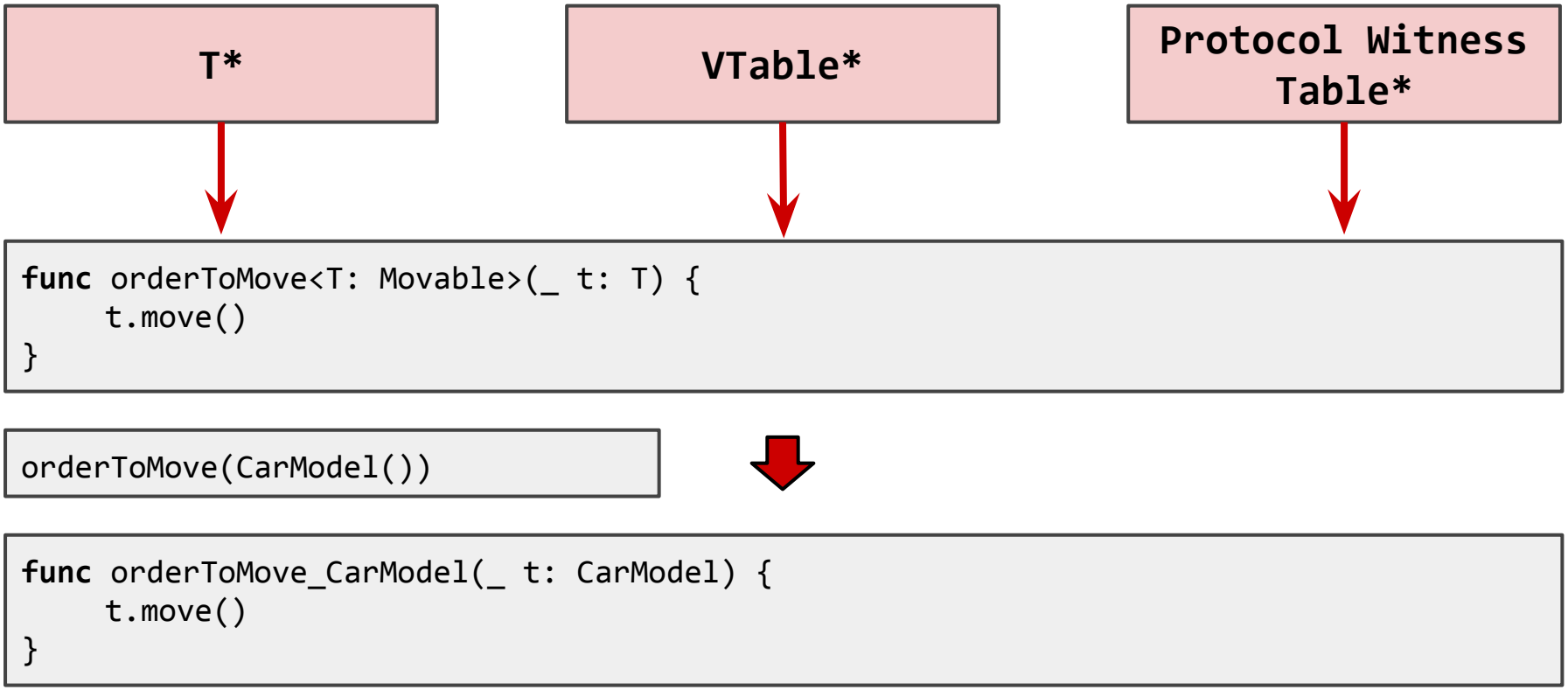
Protocol Witness Table



Protocol Witness Table



Generic specialization



Generic specialization

Type of an instance is known statically.

Type definition, generic definition and use-case are accessible.

Or:

```
@_specialize(CarModel)
func orderToMove<T: Movable>(_ t: T) {
     t.move()
}
```

Suggestion

Do not forget about: `final`, `private` and optimize whole modules.

Always provide as much static information as possible.

inga.roksana.rueb@gmail.com