Modular

Beyond RAII: Implementing Linear / Explicitly Destroyed Types in Vale and Mojo

Evan Ovadia

Can you spot the problem with these comments?

// Remember to call .join() or .detach() on this thread before it goes out of scope.

// The caller is responsible for calling thePromise.set_value(...)

// Remember to decelerate the car before you stop driving.

// Before the entity is destroyed, remove its ID from the location map.

Can you spot the problem with these comments?

// Remember to call .join() or .detach() on this thread before it goes out of scope.

// The caller is responsible for calling thePromise.set_value(...)

// Remember to decelerate the car before you stop driving.

// Before the entity is destroyed, remove its ID from the location map.

Problem: they rely on us to remember to do something!

In this talk

01	What's a linear type?
02	How it prevents bugs
03	Linear types' hidden superpower
04	Mojo implementation, in the CheckLifetimes MLIR pass
05	Vale implementation, including conditionally linear types

What's "linear"?

Usual definition:
A linear object must eventually be consumed, exactly once.

My definition:
A linear object can't just go out of scope, you must eventually explicitly destroy it in a *specific* way.

Example:
join() or detach() a thread

If you own a std::thread, either:

- Call .join() on it
- Call .detach() on it

...before it goes out of scope.

If you forget, your program crashes.

Partial solution: std::jthread

(C++)

```
void foo() {
  auto t = std::thread{...};
  ...
  // bug: t goes out of scope, we haven't
  // called .join() or .detach(), so it
  // calls std::terminate()
}
```

Example:
join() or detach() a thread

If you own a Thread, either:

- Call .join() on it
- Call .detach() on it

...before it goes out of scope.

Mojo can check this at compile time.

A struct with @explicit_destroy is never automatically deleted.

(Mojo)

```
fn foo():
  t = Thread(...)
  ...
  Error: Can't delete `t`: Must call join() or detach()

@explicit_destroy(
    "Must call join() or detach()")
struct Thread:
  ...
  fn join(owned self):
    ...
    destroy self
  fn detach(owned self):
    ...
    destroy self
```

Example:
join() or detach() a thread

One must either:

- Call a method that takes `owned self`
- Postpone by moving it, either:
  - into another function:

    `someList.append(t^)`
  - to the caller via a return:

    `return t^`

  (but will still have the same rules)

(Mojo)

```
fn foo():
  t = Thread(...)
  ...
  Error: Can't delete `t`: Must call join() or detach()

@explicit_destroy(
    "Must call join() or detach()")
struct Thread:
  ...
  fn join(owned self):
    ...
    destroy self
  fn detach(owned self):
    ...
    destroy self
```

Example:
join() or detach() a thread

One must either:

- Call a method that takes `owned self`
- Postpone by moving it, either:
  - into another function:

    `someList.append(t^)`
  - to the caller via a return:

    `return t^`

  (but will still have the same rules)

(Mojo)

```
fn foo():
  t = Thread(...)
  ...
  t^.detach()

@explicit_destroy(
    "Must call join() or detach()")
struct Thread:
  ...
  fn join(owned self):
    ...
    destroy self
  fn detach(owned self):
    ...
    destroy self
```

Example:
join() or detach() a thread

One must either:

- Call a method that takes `owned self`
- Postpone by moving it, either:
  - into another function:

    `someList.append(t^)`
  - to the caller via a return:

    `return t^`

  (but will still have the same rules)

(Mojo)

```
fn foo(inout threads: List[Thread]):
  t = Thread(...)
  ...
  threads.append(t^)

@explicit_destroy(
    "Must call join() or detach()")
struct Thread:
  ...
  fn join(owned self):
    ...
    destroy self
  fn detach(owned self):
    ...
    destroy self
```

Example:
join() or detach() a thread

One must either:

- Call a method that takes `owned self`
- Postpone by moving it, either:
  - into another function:

    `someList.append(t^)`
  - to the caller via a return:

    `return t^`

  (but will still have the same rules)

(Mojo)

```
fn foo() -> Thread:
  t = Thread(...)
  ...
  return t^

@explicit_destroy(
    "Must call join() or detach()")
struct Thread:
  ...
  fn join(owned self):
    ...
    destroy self
  fn detach(owned self):
    ...
    destroy self
```

Example: join() or detach() a thread

Partial solution: std::jthread, which automatically calls .join() when it goes out of scope.

The problem: it went out of scope too early, and serialized our loop!

(C++)

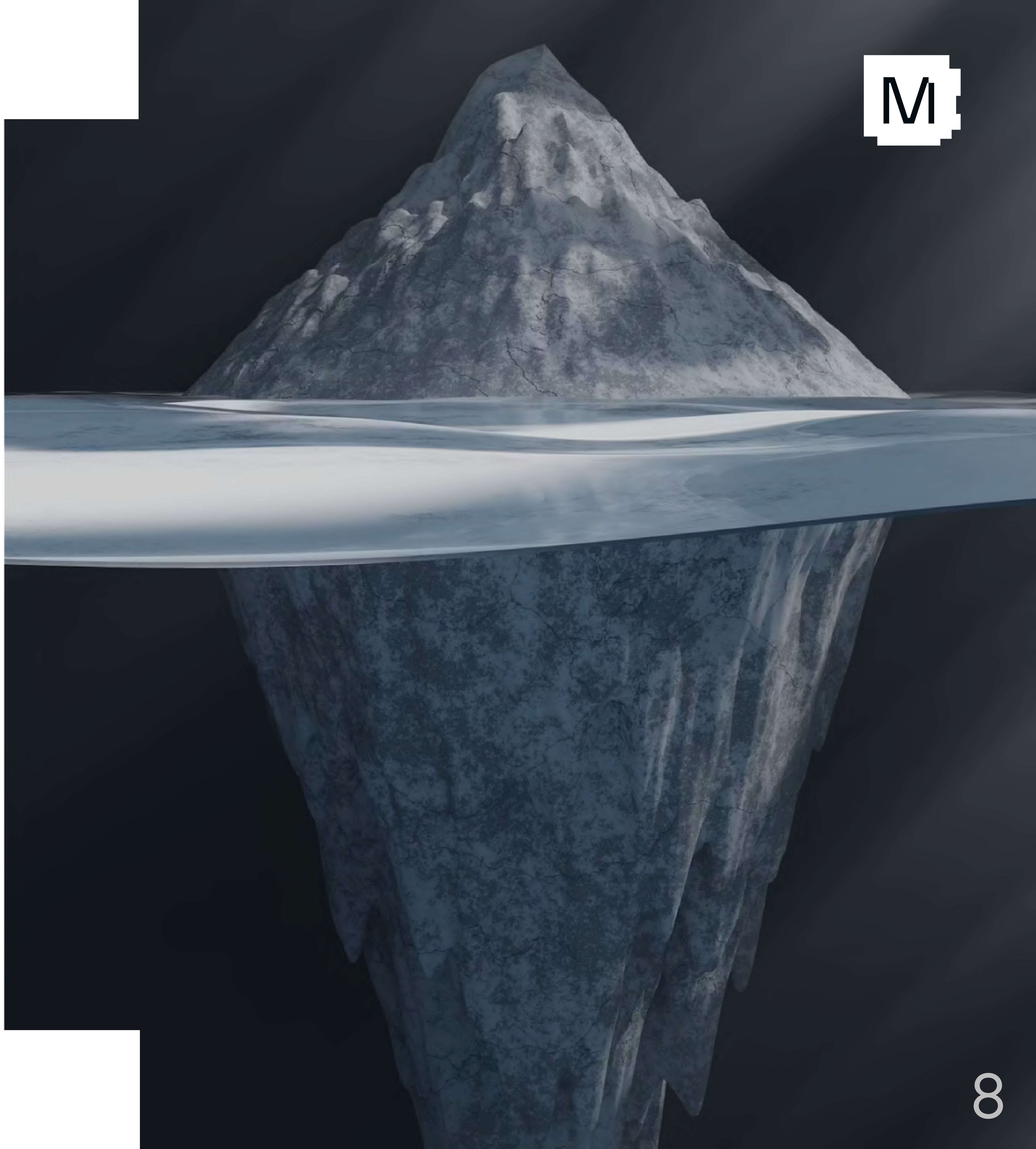
```
void foo() {
  std::vector<std::jthread> threads;

  // Parallel, hopefully
  for (int i = 0; i < 10; i++) {
    auto t = std::jthread{...};
    // Forgot to add to list!
  }

  // All threads' destructors will join
}
```

Linear types help us:

- .join() or .detach() a thread
- .set_value(result) on a promise
- .get() a future
- Decelerate before you stop driving
- Keep a rocket booster firing
- Handle failed requests
- Prevent "hanging" database rows
- Prevent "handle leaks", orphan nodes
- Ensure another thread handles a message
- Prevent inconsistent state
- Solve lookup-after-remove
- "Linear static reference counting"
- Get an error from close(fd)
- and more!



Example: Forgotten promises

One should put a value in the std::promise so that the other thread can see it.

If we forget to call set_value, then the other thread won't work.

(C++)

```
void foo() {
  std::promise<Result> p = ...
  ...
  // bug: p goes out of scope, we haven't
  // called p.set_value(result), receiving
  // thread has a problem.
}
```

Example:
Forgotten promises

Mojo can check this at compile time.

```
(Mojo)
fn foo():
  p: Promise<Result> = ...
  ...
  Error: Can't delete `p`: Use set_value()
```

Example: Forgotten promises

Mojo can check this at compile time.

A struct with @explicit_destroy is never automatically deleted.

```
                    (Mojo)
fn foo():
  p: Promise<Result> = ...
  ...
  Error: Can't delete `p`: Use set_value()

@explicit_destroy("Use set_value")
struct Promise[T]:
  ...
  fn set_value(owned self, value: T):
    ...
    destroy self
```

Example: Forgotten promises

One must either:

- Call a method that takes `owned self`
- Postpone by moving it, either:
  - into another function:

    `someList.append(t^)`
  - to the caller via a return:

    `return t^`

  (but will still have the same rules)

(Mojo)

```
fn foo():
  p: Promise<Result> = ...
  ...
  Error: Can't delete `p`: Use set_value()

@explicit_destroy("Use set_value")
struct Promise[T]:
  ...
  fn set_value(owned self, value: T):
    ...
    destroy self
```

Example:
Forgotten promises

One must either:

- Call a method that takes `owned self`
- Postpone by moving it, either:
  - into another function:

    `someList.append(t^)`
  - to the caller via a return:

    `return t^`

(but will still have the same rules)

```
                  (Mojo)
fn foo():
  p: Promise<Result> = ...
  ...
  p^.set_value(        )

@explicit_destroy("Use set_value")
struct Promise[T]:
  ...
  fn set_value(owned self, value: T):
    ...
    destroy self
```

Example: Forgotten promises

One must either:

- Call a method that takes `owned self`
- Postpone by moving it, either:
  - into another function:
    `someList.append(t^)`
  - to the caller via a return:
    `return t^`

(but will still have the same rules)

```
                    (Mojo)
fn foo():
  p: Promise<Result> = ...
  myResult = ...
  p^.set_value(myResult)

@explicit_destroy("Use set_value")
struct Promise[T]:
  ...
  fn set_value(owned self, value: T):
    ...
    destroy self
```

Example: Dropped futures

A container might have an important linear type in it.

We should extract it before destroying the container.

```
(Mojo)
fn foo():
  f: Future[ImportantLinearThing] = ...
  ...
```

Error: Can't delete `f`: Use get()

Example: Dropped futures

A container might have an important linear type in it.

We should extract it before destroying the container.

Mojo can check this at compile time.

A struct with @explicit_destroy is never automatically deleted.

```
                    (Mojo)
fn foo():
  f: Future[ImportantLinearThing] = ...
  ...
  Error: Can't delete `f`: Use get()

@explicit_destroy("Use get()")
struct Future[T]:
  ...
  fn get(owned self) -> T:
    self.wait()
    v = self.value^
    destroy self
    return v^
```

Example: Dropped futures

A container might have an important linear type in it.

We should extract it before destroying the container.

Mojo can check this at compile time.

A struct with @explicit_destroy is never automatically deleted.

```
                    (Mojo)
fn foo():
  f: Future[ImportantLinearThing] = ...
  ...
  thing = f^.get()

@explicit_destroy("Use get()")
struct Future[T]:
  ...
  fn get(owned self) -> T:
    self.wait()
    v = self.value^
    destroy self
    return v^
```

A pattern emerges

They ensured we eventually made a decision:

- `t^.join()`
- `t^.detach()`

They ensured we eventually calculated and gave a value:

- `p^.set_value(myResult)`

They ensured we eventually took a value:

- `value = f^.get()`

With linear types, you **control the future.**

You can craft a linear type's methods (which take `owned self`) to help enforce what will eventually happen:

Multiple methods: a decision to eventually make.

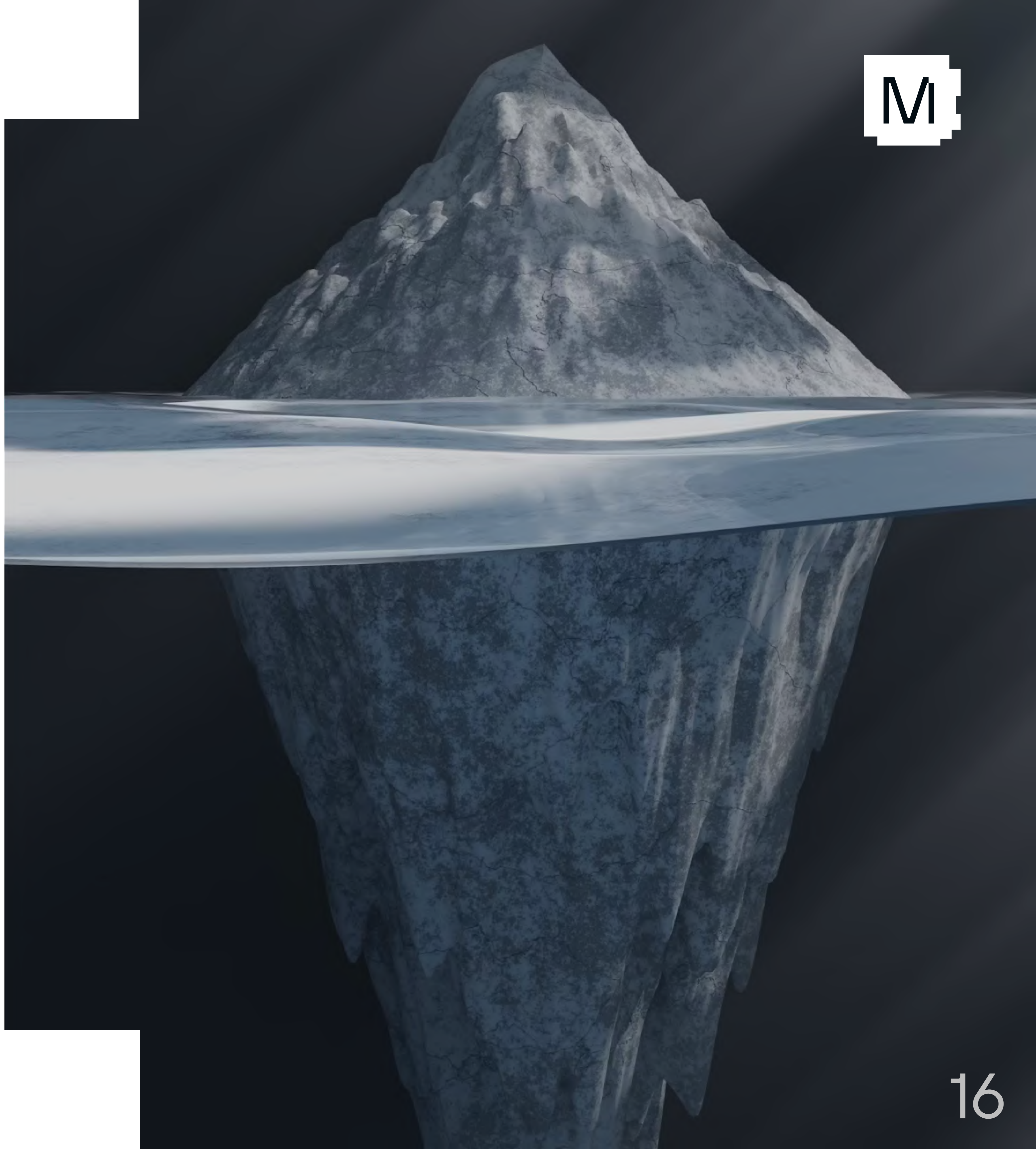
Arguments: data to eventually calculate and give.

Return values: data to eventually take.

You can combine these!

Linear types help us:

- .join() or .detach() a thread
- .get() a future
- .set_value(result) on a promise
- Decelerate before you stop driving
- Keep a rocket booster firing
- Handle failed requests
- Prevent "hanging" database rows
- Prevent "handle leaks", orphan nodes
- Ensure another thread handles a message
- Prevent inconsistent state
- Solve lookup-after-remove
- "Linear static reference counting"
- Get an error from close(fd)
- and more!



Example: Prevent inconsistent state

Add an entity to the LiveEntityList to get some LiveEntityHandles.

These handles can go in e.g.:

- A location-to-entity-handle map
- A faction-to-entity-handle map

Remove an Entity from the LiveEntityList by giving back the two LiveEntityHandles.

```
# Can only be created by LiveEntityList
@explicit_destroy(
    "Use LiveEntityList's remove")
struct LiveEntityHandle:
  var index: Int

struct LiveEntityList:
  var entities: List[Entity]

  fn add(owned e: Entity) ->
      (LiveEntityHandle, LiveEntityHandle):
    …

  fn remove(
      owned h1: LiveEntityHandle,
      owned h2: LiveEntityHandle) -> Entity:
    …
```

Example: Prevent inconsistent state

Add an entity to the LiveEntityList to get some LiveEntityHandles.

These handles can go in e.g.:

- A location-to-entity-handle map
- A faction-to-entity-handle map

Remove an Entity from the LiveEntityList by giving back the two LiveEntityHandles.

```
# Can only be created by LiveEntityList
@explicit_destroy(
    "Use LiveEntityList's remove")
struct LiveEntityHandle:
  var index: Int

struct LiveEntityList:
  var entities: List[Entity]

  fn add(owned e: Entity) ->
      (LiveEntityHandle,  # for location map
       LiveEntityHandle): # for faction map
    …
  fn remove(
      owned h1: LiveEntityHandle,
      owned h2: LiveEntityHandle) -> Entity:
    …
```

Example: Prevent inconsistent state

- If you have a LiveEntityHandle, you know it's still in the LiveEntityList.
- "Dangling" LiveEntityHandles are impossible!
- Must take the handles from the maps before you can remove the entity from the LiveEntityList
- Maps can never get out of sync!

```
# Can only be created by LiveEntityList
@explicit_destroy(
    "Use LiveEntityList's remove")
struct LiveEntityHandle:
  var index: Int

struct LiveEntityList:
  var entities: List[Entity]

  fn add(owned e: Entity) ->
      (LiveEntityHandle,  # for location map
       LiveEntityHandle): # for faction map
    …
  fn remove(
      owned h1: LiveEntityHandle,
      owned h2: LiveEntityHandle) -> Entity:
    …
```

Example: Prevent inconsistent state

- If you have a LiveEntityHandle, you know it's still in the LiveEntityList.
- "Dangling" LiveEntityHandles are impossible!
- Must take the handles from the maps before you can remove the entity from the LiveEntityList
- Maps can never get out of sync!

"Non-scoped borrowing"?
"Linear compile-time ref-counting"?

```
# Can only be created by LiveEntityList
@explicit_destroy(
    "Use LiveEntityList's remove")
struct LiveEntityHandle:
  var index: Int

struct LiveEntityList:
  var entities: List[Entity]

  fn add(owned e: Entity) ->
      (LiveEntityHandle,  # for location map
       LiveEntityHandle): # for faction map
    …
  fn remove(
      owned h1: LiveEntityHandle,
      owned h2: LiveEntityHandle) -> Entity:
    …
```

Context: ASAP Destruction

Normal structs in Mojo use "ASAP destruction".

The `__del__` call is inserted as early as possible.

```
struct Ship:
  var hp: Int

fn main():
  ship = Ship(42)
  x = ship.hp
  print(x)
```

Context: ASAP Destruction

Normal structs in Mojo use "ASAP destruction".

The `__del__` call is inserted as early as possible.

```
struct Ship:
  var hp: Int

fn main():
  ship = Ship(42)
  x = ship.hp
  ship^.__del__()
  print(x)
```

CheckLifetimes MLIR Pass

CheckLifetimes pass will:

- Finds lifetime starts and ends.
- Insert any __del__ destructor calls.

CheckLifetimes MLIR Pass

```
lit.func @"main()"() -> !kgen.none {
  %x = decl "x" …
  %ship = decl "ship" …
  %0 = constant 42 …
  %1 = call @"Ship::__init__"(%ship, %0) …
  %2 = ger %ship[hp] …
  %3 = load %2 …
       store %3, %x …
  %5 = immut %x …
  %7 = call @"print[Int]"(%5) …
}
```

```
fn main():



  ship = Ship(42)


  x = ship.hp

  print(x)
```

CheckLifetimes MLIR Pass

```
lit.func @"main()"() -> !kgen.none {
  %x = decl "x" …
  %ship = decl "ship" …
  %0 = constant 42 …
  %1 = call @"Ship::__init__"(%ship, %0) {start ship} …
  %2 = ger %ship[hp] {use ship} …
  %3 = load %2 {use ship} …
       store %3, %x {start x} …
  %5 = immut %x {use x} …
  %7 = call @"print[Int]"(%5) {use x} …
}
```

```
fn main():
  ship = Ship(42)
  x = ship.hp
  print(x)
```

CheckLifetimes MLIR Pass

```
lit.func @"main()"() -> !kgen.none {
  %x = decl "x" …
  %ship = decl "ship" …
  %0 = constant 42 …
  %1 = call @"Ship::__init__"(%ship, %0) {start ship} …
  %2 = ger %ship[hp] {use ship} …
  %3 = load %2 {use ship} …
       store %3, %x {start x} …
  %5 = immut %x {use x} …
→ %7 = call @"print[Int]"(%5) {use x} …
}
```

```
fn main():



  ship = Ship(42)


  x = ship.hp

  print(x)
```

CheckLifetimes MLIR Pass

```
lit.func @"main()"() -> !kgen.none {
  %x = decl "x" …
  %ship = decl "ship" …
  %0 = constant 42 …
  %1 = call @"Ship::__init__"(%ship, %0) {start ship} …
  %2 = ger %ship[hp] {use ship} …
  %3 = load %2 {use ship} …
→      store %3, %x {start x} …
  %5 = immut %x {use x} …
  %7 = call @"print[Int]"(%5) {use x} …
       lifetime.end %x …
}
```

```
fn main():
  ship = Ship(42)
  x = ship.hp
  print(x)
```

CheckLifetimes MLIR Pass

```
lit.func @"main()"() -> !kgen.none {
  %x = decl "x" …
  %ship = decl "ship" …
  %0 = constant 42 …
  %1 = call @"Ship::__init__"(%ship, %0) {start ship} …
  %2 = ger %ship[hp] {use ship} …
→ %3 = load %2 {use ship} …
       lifetime.start %x
       store %3, %x {start x} …
  %5 = immut %x {use x} …
  %7 = call @"print[Int]"(%5) {use x} …
       lifetime.end %x …
}
```

```
fn main():
  ship = Ship(42)
  x = ship.hp
  print(x)
```

CheckLifetimes MLIR Pass

```
lit.func @"main()"() -> !kgen.none {
  %x = decl "x" …
  %ship = decl "ship" …
  %0 = constant 42 …
  %1 = call @"Ship::__init__"(%ship, %0) {start ship} …
  %2 = ger %ship[hp] {use ship} …
→ %3 = load %2 {use ship} …
       lifetime.end %ship
       lifetime.start %x
       store %3, %x {start x} …
  %5 = immut %x {use x} …
  %7 = call @"print[Int]"(%5) {use x} …
       lifetime.end %x …
}
```

```
fn main():
  ship = Ship(42)
  x = ship.hp
  print(x)
```

CheckLifetimes MLIR Pass

```
lit.func @"main()"() -> !kgen.none {
  %x = decl "x" …
  %ship = decl "ship" …
  %0 = constant 42 …
  %1 = call @"Ship::__init__"(%ship, %0) {start ship} …
  %2 = ger %ship[hp] {use ship} …
  %3 = load %2 {use ship} …
  %8 = call @"Ship::__del__"(%ship)
       lifetime.end %ship
       lifetime.start %x
       store %3, %x {start x} …
  %5 = immut %x {use x} …
  %7 = call @"print[Int]"(%5) {use x} …
       lifetime.end %x …
}
```

```
fn main():

  ship = Ship(42)

  x = ship.hp

  print(x)
```

CheckLifetimes MLIR Pass

```
lit.func @"main()"() -> !kgen.none {
  %x = decl "x" …
  %ship = decl "ship" …
  %0 = constant 42 …
       lifetime.start %ship
  %1 = call @"Ship::__init__"(%ship, %0) {start ship} …
  %2 = ger %ship[hp] {use ship} …
  %3 = load %2 {use ship} …
  %8 = call @"Ship::__del__"(%ship)
       lifetime.end %ship
       lifetime.start %x
       store %3, %x {start x} …
  %5 = immut %x {use x} …
  %7 = call @"print[Int]"(%5) {use x} …
       lifetime.end %x …
}
```

```
fn main():
  ship = Ship(42)
  x = ship.hp
  print(x)
```

CheckLifetimes with linear types

This is the correct code.

```
fn main():
  ship = Ship(42)
  hp = ship.hp

  landing_zone = ...
  ship^.land(landing_zone)
  print(hp)
```

CheckLifetimes with linear types

This is the correct code.

```
fn main():
  ship = Ship(42)
  hp = ship.hp

  landing_zone = ...
  ship^.land(landing_zone)
  print(hp)

@explicit_destroy("Use land()")
struct Ship:
  var hp: Int
  ...
  fn land(owned self, landing_zone: Loc):
    ...
    destroy self
```

CheckLifetimes with linear types

```
fn main():
  ship = Ship(42)
  hp = ship.hp
  Error: Can't delete `ship`: Use land()
  # landing_zone = ...
  # ship^.land(landing_zone)
  print(hp)

@explicit_destroy("Use land()")
struct Ship:
  var hp: Int
  ...
  fn land(owned self, landing_zone: Loc):
    ...
    destroy self
```

CheckLifetimes (with linear types)

```
lit.func @"main()"() -> !kgen.none {
  %x = decl "x" …
  %ship = decl "ship" …
  %0 = constant 42 …
  %1 = call @"Ship::__init__"(%ship, %0) {start ship} …
  %2 = ger %ship[hp] {use ship} …
→ %3 = load %2 {use ship} …
       lifetime.end %ship
       lifetime.begin %x
       store %3, %x {start x} …
  %5 = immut %x {use x} …
  %7 = call @"print[Int]"(%5) {use x} …
       lifetime.end %x …
}
```

```
fn main():
  ship = Ship(42)
  x = ship.hp
  print(x)
```

CheckLifetimes (with linear types)

```
lit.func @"main()"() -> !kgen.none {
  %x = decl "x" …
  %ship = decl "ship" …
  %0 = constant 42 …
  %1 = call @"Ship::__init__"(%ship, %0) {start ship} …
  %2 = ger %ship[hp] {use ship} …
  %3 = load %2 {use ship} …
       Error: Can't delete `ship`: Use land()
       lifetime.end %ship
       lifetime.begin %x
       store %3, %x {start x} …
  %5 = immut %x {use x} …
  %7 = call @"print[Int]"(%5) {use x} …
       lifetime.end %x …
}
```

```
fn main():
  ship = Ship(42)
  x = ship.hp
  print(x)
```

The Container Problem

Can't assume T has a destructor.

If T doesn't have a destructor, what does Box[T]'s destructor call?

```
struct Box[T: AnyType]:
  var value: T
  # auto-generated
  fn __del__(owned self):
    Error: Can't delete `self.value`: No __del__
    self.value^.__del__()
```

Vale's Conditionally Linear Types

Can't assume T has a destructor.

If T doesn't have a destructor, what does Box[T]'s destructor call?

```
(Vale, using Mojo-ish syntax)
struct Box[T: AnyType]:
  var value: T

  fn __del__(owned self)
      where exists T::__del__():
    self.value^.__del__()
```

Roadmap

- Basic compiler support (Done)
- Conditionally linear types
- Update standard library: Dict, List, Box, Variant, etc.
- Launch behind a compiler flag, e.g. —enable_explicit_destroy
- Get community feedback
- If it all looks good, enable by default!

Questions

Some good unanswered ones:

- Can we have linear types in C++?
- Compared to RAII
- Compared to [[nodiscard]]
- How strong of a guarantee is it?
- Where can/can't we have linear types?
- How are these linear types?

Open roles at Modular ⬆️

Linear types in C++?

Difficult in C++ because of exceptions, stack unwinding.

- Mojo doesn't have exceptions / stack unwinding, so not a problem.
- Same with Vale, no stack unwinding.
- Vale hopes/dreams: onPanic function, region-based software transactional memory

Almost there: C++ has private destructors, but std::move doesn't actually destroy its source.

More powerful than [[nodiscard]]; follows the type through the codebase, past this function.

How strong of a guarantee is this?

Answer: pretty strong.

Except:

- Memory leaks
- exit() before destroying linear types

Are there places that can't hold linear types?

Answer: Yes, occasionally.

Reference counted objects normally require a zero-arg destructor. Some possibilities:

- Just require all reference counted things to have a zero-arg destructor.
- One linear OwningRef<Thing>, multiple RefCounted<Optional<Thing>>

Globals might require a zero-arg destructor, run after main. Some possibilities:

- Just require all globals to have a zero-arg destructor.
- Explicitly initialize and destroy all globals.

A more flexible, super-powered RAII

RAII can call one destructor, with zero arguments, and no return.

Linear types can help us remember to call:

- A function with many arguments: `p^.set_value(result)`
- A function with a return value: `value = f^.get()`
- One of many valid options: `t^.join()` vs `t^.detach()`