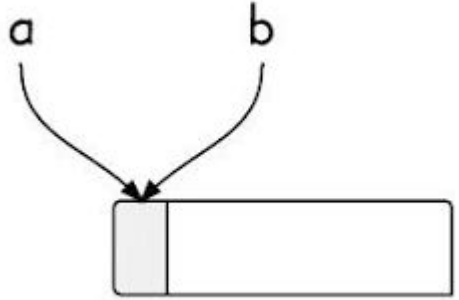
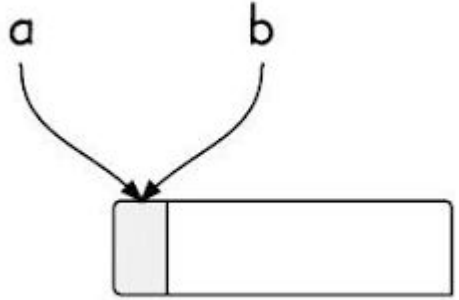
L^3: A Linear Language with Locations

Ray Tan, Eric Yang

Aliasing: two variables pointing to the same memory location



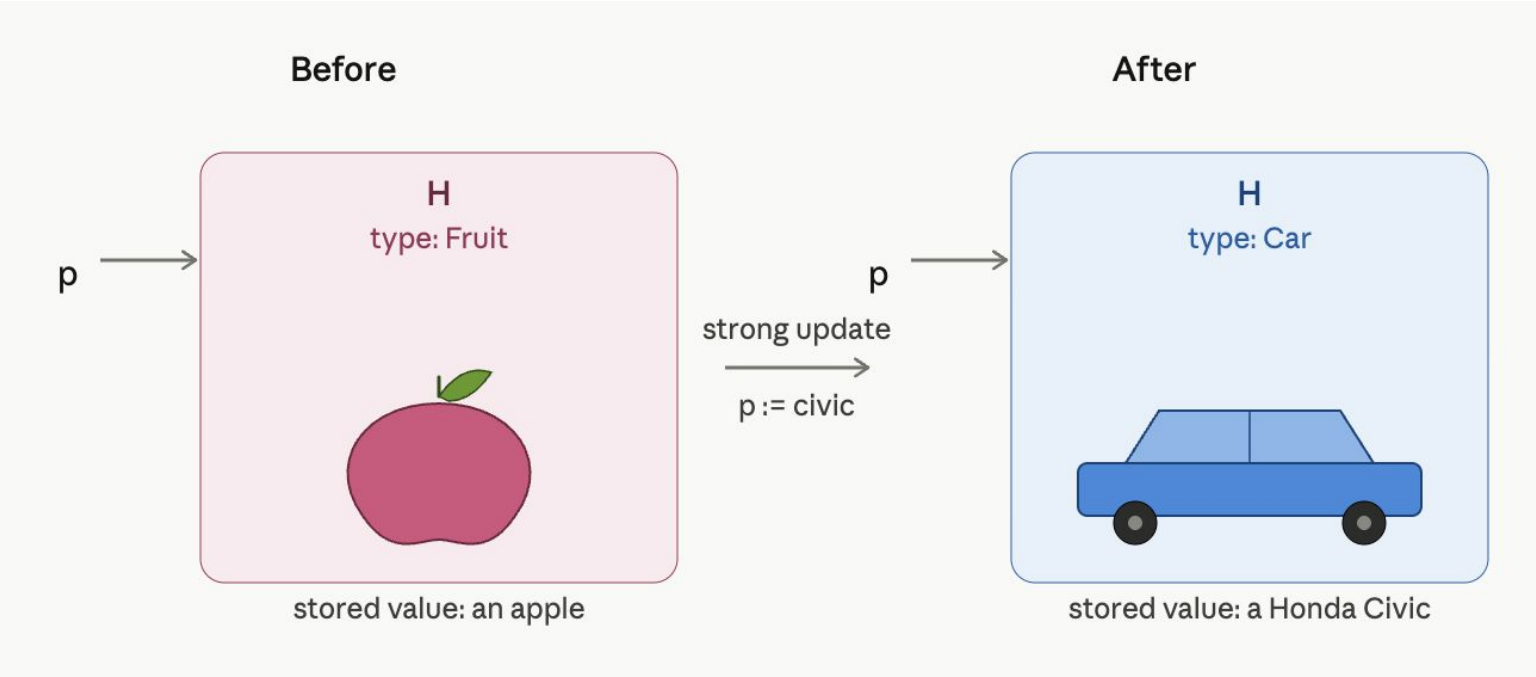
Aliasing: two variables pointing to the same memory location



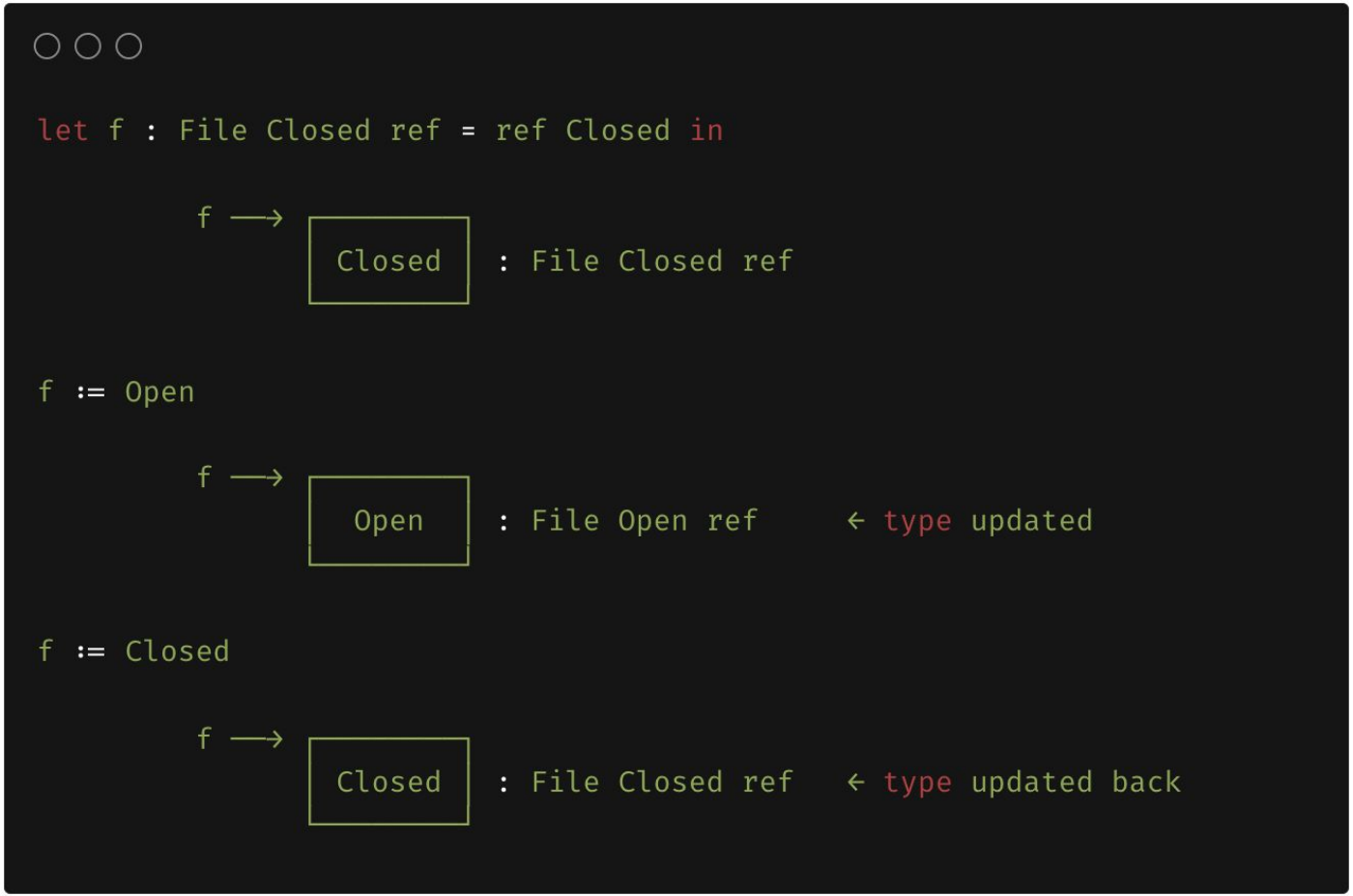

```
(* Aliasing *)

let x = ref 0 in
let y = x in                    (* y aliases x *)
y := 1;
!x                              (* returns 1 *)
```

Strong Updates: Changing a mutable object's type

Strong Updates: Changing a mutable object's type



Strong updates in action:



Why Strong Updates?

Why Strong Updates?

```
(* Without State Tracking *)

let f : File ref = ref Closed in

read(!f)
(* compiles, but may fail at runtime *)

f := Open

read(!f)
(* works *)

f := Closed

read(!f)
(* compiles, but may fail at runtime *)
```

```
(* With State Tracking *)

let f : File Closed ref = ref Closed in

read(!f)
(* compile error: file is closed *)

f := Open

read(!f)
(* works *)

f := Closed

read(!f)
(* compile error again *)
```

```
(* Without State Tracking *)

let f : File ref = ref Closed in

read(!f)
(* compiles, but may fail at runtime *)

f := Open

read(!f)
(* works *)

f := Closed

read(!f)
(* compiles, but may fail at runtime *)
```

Strong Updates with Aliasing?

Strong Updates with Aliasing?

```
fun bad(r1 : int ref, r2 : int ref) : int =
  (r1 := true;                (* int ref → bool ref *)
  !r2 + 42)
```

Strong Updates with Aliasing?

```
fun bad(r1 : int ref, r2 : int ref) : int =
  (r1 := true;                  (* int ref → bool ref *)
  !r2 + 42)                     (* true + 42 *)
```

Breaks Soundness!

```
fun bad(r1 : int ref, r2 : int ref) : int =
  (r1 := true;                    (* int ref → bool ref *)
  !r2 + 42)                       (* true + 42 *)
```

Strong Updates with Aliasing?

```
fun bad(r1 : int ref, r2 : int ref) : int =
  (r1 := true;                    (* int ref → bool ref *)
  !r2 + 42)                       (* true + 42 *)

let r = ref 0 in
bad(r, r)                         (* r1 and r2 alias *)
```

Alias Types

Core idea: Separate references into pointer and capability

`!(Ptr ρ)` — Pointer to location `ρ`

`Cap ρ τ` — Capability for location `ρ` which currently stores `τ`

Alias Types

Core idea: Separate references into pointer and capability

`!(Ptr ρ)` — Pointer to location `ρ`

`Cap ρ τ` — Capability for location `ρ` which currently stores `τ`

Allow strong updates only when capability is presented.

Alias Types

Core idea: Separate references into pointer and capability

`!(Ptr ρ)` — Pointer to location `ρ`

`Cap ρ τ` — Capability for location `ρ` which currently stores `τ`

Allow strong updates only when capability is presented.

Capabilities are linear — only one reference at a time.

Alias Types

Core idea: Separate references into pointer and capability

`!(Ptr ρ)` — Pointer to location `ρ`

`Cap ρ τ` — Capability for location `ρ` which currently stores `τ`

Allow strong updates only when capability is presented.

Capabilities are linear — only one reference at a time.

What would a strong update look like under these rules?

Strong Update Example with Alias Types

```
fun f (c1 : Cap ρ τ1) (p : !(Ptr ρ) ⊗ !(Ptr ρ)) (v : τ2) =
    let val (x, y) = p
        val (c2, z) = swap c1 x v in
        (c2, y, z)
    end
```

Perform a strong update to location ρ and replace it with v.

Strong Update Example with Alias Types

```
fun f (c1 : Cap ρ τ1) (p : !(Ptr ρ) ⊗ !(Ptr ρ)) (v : τ2) =
    let val (x, y) = p
        val (c2, z) = swap c1 x v in
        (c2, y, z)
    end
```

Perform a strong update to location ρ and replace it with v.

Strong Update Example with Alias Types

```
fun f (c1 : Cap ρ τ1) (p : !(Ptr ρ) ⊗ !(Ptr ρ)) (v : τ2) =
    let val (x, y) = p
        val (c2, z) = swap c1 x v in
        (c2, y, z)
    end
```

Perform a strong update to location ρ and replace it with v.

Strong Update Example with Alias Types

```
fun f (c1 : Cap ρ τ1) (p : !(Ptr ρ) ⊗ !(Ptr ρ)) (v : τ2) =
    let val (x, y) = p
        val (c2, z) = swap c1 x v in
        (c2, y, z)
    end
```

Perform a strong update to location ρ and replace it with v.

Strong Update Example with Alias Types

```
fun f (c1 : Cap ρ τ1) (p : !(Ptr ρ) ⊗ !(Ptr ρ)) (v : τ2) =
    let val (x, y) = p
        val (c2, z) = swap c1 x v in
        (c2, y, z)
    end
```

Unpack the pair of pointers in `p`

Strong Update Example with Alias Types

```
fun f (c1 : Cap ρ τ1) (p : !(Ptr ρ) ⊗ !(Ptr ρ)) (v : τ2) =
    let val (x, y) = p
        val (c2, z) = swap c1 x v in
        (c2, y, z)
    end
```

Using capability `c1`, replace contents of location `x` with value `v`

Strong Update Example with Alias Types

```
fun f (c1 : Cap ρ τ1) (p : !(Ptr ρ) ⊗ !(Ptr ρ)) (v : τ2) =
    let val (x, y) = p
        val (c2, z) = swap c1 x v in
        (c2, y, z)
    end
```

Return a new capability `c2`, a sharable pointer `y`, and the old value `z`

The L^3 Language

L^3: Linear Language with Locations

The L^3 Language

L^3: Linear Language with Locations

- Based on Lambda Calculus

The L^3 Language

L^3: Linear Language with Locations

- Based on Lambda Calculus
- Uses Alias Types

The L^3 Language

L^3: Linear Language with Locations

- Based on Lambda Calculus
- Uses Alias Types
- Uses a store which contains mutable boxes

The L^3 Language

L^3: Linear Language with Locations

- Based on Lambda Calculus
- Uses Alias Types
- Uses a store which contains mutable boxes

```
σ ::= {ℓ1 ↦ v1, ..., ℓn ↦ vn}
```

The L^3 Language

L^3: Linear Language with Locations

- Based on Lambda Calculus
- Uses Alias Types
- Uses a store which contains mutable boxes

σ ::= {ℓ1 ↦ v1, ..., ℓn ↦ vn}

LocConsts ℓ ∈ *LocConsts*

LocVars ρ ∈ *LocVars*

Locs η ::= ℓ | ρ

L^3 Semantics

Types τ ::= **I** | τ1 ⊗ τ2 | τ1 ⊸ τ2 | !τ | Ptr η | Cap η τ | ∀ρ.τ | ∃ρ.τ

L^3 Semantics

Types τ ::= **I** | τ1 ⊗ τ2 | τ1 ⊸ τ2 | !τ | Ptr η | Cap η τ | ∀ρ.τ | ∃ρ.τ

Exprs e ::= * | let * = e1 in e2 |

Unit & Unit constructor

L^3 Semantics

Types τ ::= **𝟙** | τ1 ⊗ τ2 | τ1 ⊸ τ2 | !τ | Ptr η | Cap η τ | ∀ρ.τ | ∃ρ.τ

Exprs e ::= * | let * = e1 in e2 |

⟨e1, e2⟩ | let ⟨x1, x2⟩ = e1 in e2 |

Tensor Pair & Constructor

L^3 Semantics

Types τ ::= **𝟙** | τ1 ⊗ τ2 | τ1 ⊸ τ2 | !τ | Ptr η | Cap η τ | ∀ρ.τ | ∃ρ.τ

Exprs e ::= * | let * = e1 in e2 |

⟨e1, e2⟩ | let ⟨x1, x2⟩ = e1 in e2 |

x | λx. e | e1 e2 |

Lambda calculus

L^3 Semantics

Types τ ::= `𝟙 | τ1 ⊗ τ2 | τ1 ⊸ τ2 | !τ | Ptr η | Cap η τ | ∀ρ.τ | ∃ρ.τ`

Exprs e ::= `* | let * = e1 in e2 |`

`⟨e1, e2⟩ | let ⟨x1, x2⟩ = e1 in e2 |`

`x | λx. e | e1 e2 |`

`!v | let !x = e1 in e2 |`

“of course”/Unrestricted value & Constructor

L^3 Semantics

Types τ ::= `I | τ1 ⊗ τ2 | τ1 ⊸ τ2 | !τ | Ptr η | Cap η τ | ∀ρ.τ | ∃ρ.τ`

Exprs e ::= `* | let * = e1 in e2 |`

`⟨e1, e2⟩ | let ⟨x1, x2⟩ = e1 in e2 |`

`x | λx. e | e1 e2 |`

`!v | let !x = e1 in e2 | dupl e | drop e |`

`ptr ρ | cap | new e | free e | swap e1 e2 e3 |`

Linear operations and alias types

L^3 Semantics

Types τ ::= `𝐈 | τ1 ⊗ τ2 | τ1 ⊸ τ2 | !τ | Ptr η | Cap η τ | ∀ρ.τ | ∃ρ.τ`

Exprs e ::= `* | let * = e1 in e2 |`

`⟨e1, e2⟩ | let ⟨x1, x2⟩ = e1 in e2 |`

`x | λx. e | e1 e2 |`

`!v | let !x = e1 in e2 | dupl e | drop e |`

`ptr ρ | cap | new e | free e | swap e1 e2 e3 |`

`Λρ. e | e[η] |`

Location-level universal function and application

L^3 Semantics

Types τ ::= `𝟭 | τ1 ⊗ τ2 | τ1 ⊸ τ2 | !τ | Ptr η | Cap η τ | ∀ρ.τ | ∃ρ.τ`

Exprs e ::= `* | let * = e1 in e2 |`

`⟨e1, e2⟩ | let ⟨x1, x2⟩ = e1 in e2 |`

`x | λx. e | e1 e2 |`

`!v | let !x = e1 in e2 | dupl e | drop e |`

`ptr ρ | cap | new e | free e | swap e1 e2 e3 |`

`Λρ. e | e[η] | ⌜η, e⌝ | let ⌜ρ, x⌝ = e1 in e2`

Existential location package & Constructor

L^3 Semantics

Types τ ::= **𝐈** | τ1 ⊗ τ2 | τ1 ⊸ τ2 | !τ | Ptr η | Cap η τ | ∀ρ.τ | ∃ρ.τ

Exprs e ::= * | let * = e1 in e2 |

⟨e1, e2⟩ | let ⟨x1, x2⟩ = e1 in e2 |

x | λx. e | e1 e2 |

!v | let !x = e1 in e2 | dupl e | drop e |

ptr ρ | cap | new e | free e | swap e1 e2 e3 |

Λρ. e | e[η] | ⌜η, e⌝ | let ⌜ρ, x⌝ = e1 in e2

Values v ::= * | ⟨v1, v2⟩ | x | λx. e | !v | ptr ℓ | cap | Λρ. e |

⌜η, v⌝

L^3 Example: Linear Reference Swap

LRef $\tau \equiv$ $\exists \rho.(\text{Cap } \rho\ \tau \otimes \text{!Ptr } \rho)$

A linear reference holding a pointer and capability

L^3 Example: Linear Reference Swap

LRef $\tau \equiv$ $\exists \rho.(\texttt{Cap}\ \rho\ \tau \otimes\ !\texttt{Ptr}\ \rho)$

How do we update the value at the location of this linear reference?

A linear reference holding a pointer and capability

L^3 Example: Linear Reference Swap

LRef τ ≡ `∃ρ.(Cap ρ τ ⊗ !Ptr ρ)`

lrswap ≡ `λr:LRef τ. λx:τ′.`

Two inputs: a LRef and the new value

L^3 Example: Linear Reference Swap

LRef τ ≡ `∃ρ.(Cap ρ τ ⊗ !Ptr ρ)`

lrswap ≡ `λr:LRef τ. λx:τ'.`

`let ⌜ρ, cp⌝ = r in` `# cp: Cap ρ τ ⊗ !Ptr ρ`

Unpack the LRef `r` into an existential location package

L^3 Example: Linear Reference Swap

LRef τ ≡ `∃ρ.(Cap ρ τ ⊗ !Ptr ρ)`

lrswap ≡ `λr:LRef τ. λx:τ'.`

```
let ⌜ρ, cp⌝ = r in                    # cp: Cap ρ τ ⊗ !Ptr ρ
let ⟨c0, p0⟩ = cp in                  # c0: Cap ρ τ, p0: !Ptr ρ
```

Unpack the pair `cp` into the capability `c0` and shareable pointer `p0`

L^3 Example: Linear Reference Swap

LRef τ ≡ `∃ρ.(Cap ρ τ ⊗ !Ptr ρ)`

lrswap ≡ `λr:LRef τ. λx:τ'.`

```
let ⌜ρ, cp⌝ = r in                    # cp: Cap ρ τ ⊗ !Ptr ρ
let ⟨c0, p0⟩ = cp in                  # c0: Cap ρ τ, p0: !Ptr ρ
let ⟨p1, p2⟩ = dupl p0 in             # p1: !Ptr ρ, p2: !Ptr ρ
```

Duplicate the pointer `p0` into `p1` and `p2`

L^3 Example: Linear Reference Swap

LRef τ ≡ `∃ρ.(Cap ρ τ ⊗ !Ptr ρ)`

lrswap ≡ `λr:LRef τ. λx:τ'.`

```
let ⌜ρ, cp⌝ = r in                          # cp: Cap ρ τ ⊗ !Ptr ρ
let ⟨c0, p0⟩ = cp in                        # c0: Cap ρ τ, p0: !Ptr ρ
let ⟨p1, p2⟩ = dupl p0 in                   # p1: !Ptr ρ, p2: !Ptr ρ
```

Why do we do this? (Hint: What does `swap` consume?)

Duplicate the pointer `p0` into `p1` and `p2`

L^3 Example: Linear Reference Swap

LRef τ ≡ `∃ρ.(Cap ρ τ ⊗ !Ptr ρ)`

lrswap ≡ `λr:LRef τ. λx:τ'.`

```
let ⌜ρ, cp⌝ = r in                    # cp: Cap ρ τ ⊗ !Ptr ρ
let ⟨c0, p0⟩ = cp in                  # c0: Cap ρ τ, p0: !Ptr ρ
let ⟨p1, p2⟩ = dupl p0 in             # p1: !Ptr ρ, p2: !Ptr ρ
let !p'2 = p2 in                      # p'2: Ptr ρ
```

Revert `p2` into a regular pointer `p'2` (to be consumed by `swap`)

L^3 Example: Linear Reference Swap

LRef τ ≡ `∃ρ.(Cap ρ τ ⊗ !Ptr ρ)`

lrswap ≡ `λr:LRef τ. λx:τ′.`

```
let ⌜ρ, cp⌝ = r in                    # cp: Cap ρ τ ⊗ !Ptr ρ
let ⟨c0, p0⟩ = cp in                  # c0: Cap ρ τ, p0: !Ptr ρ
let ⟨p1, p2⟩ = dupl p0 in             # p1: !Ptr ρ, p2: !Ptr ρ
let !p′2 = p2 in                      # p′2: Ptr ρ
let ⟨c1, y⟩ = swap c0 p′2 x in        # c1: Cap ρ τ′, y: τ
```

Perform the swap, consuming capability `c0`, pointer `p′2` and producing new capability `c1` and old value `y`

L^3 Example: Linear Reference Swap

LRef τ ≡ `∃ρ.(Cap ρ τ ⊗ !Ptr ρ)`

lrswap ≡

```
λr:LRef τ. λx:τ'.
  let ⌜ρ, cp⌝ = r in                    # cp: Cap ρ τ ⊗ !Ptr ρ
  let ⟨c0, p0⟩ = cp in                  # c0: Cap ρ τ, p0: !Ptr ρ
  let ⟨p1, p2⟩ = dupl p0 in             # p1: !Ptr ρ, p2: !Ptr ρ
  let !p'2 = p2 in                      # p'2: Ptr ρ
  let ⟨c1, y⟩ = swap c0 p'2 x in        # c1: Cap ρ τ', y: τ
  ⟨ ⌜ρ, ⟨c1, p1⟩⌝ , y⟩
```

Return a pair containing the new LRef, and the old value `y`

Q&A