Polymorphic Reachability Types and Effects: An Overview

Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs

Guannan Wei
INRIA/ENS, Tufts University
LIPN, Université Sorbonne Paris Nord – Mar 6, 2025

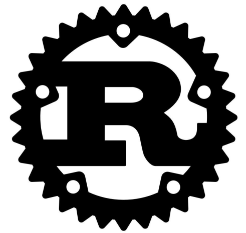
Previously at Purdue University
with Oliver Bračevac, Songlin Jia, David Deng, Siyuan He, Yuyan Bao, Tiark Rompf

Motivation

Memory safety, *thread safety, performance, ...*

Secret sauce: ownership types

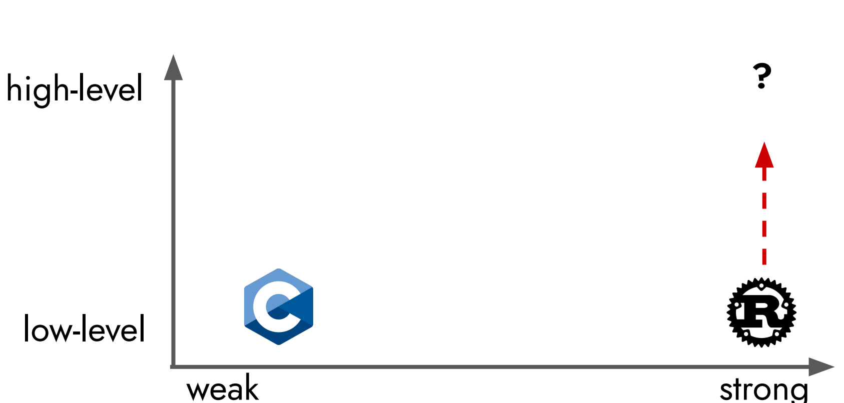
[Clarke et al., OOPSLA 98, Noble et al. ECOOP 98]

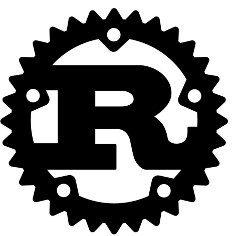


Rust most admired language, Stack Overflow survey says

Stack Overflow 2023 Developer Survey finds that JavaScript and Python are the most used and most desired languages, but they fall far short of Rust in satisfying their users.

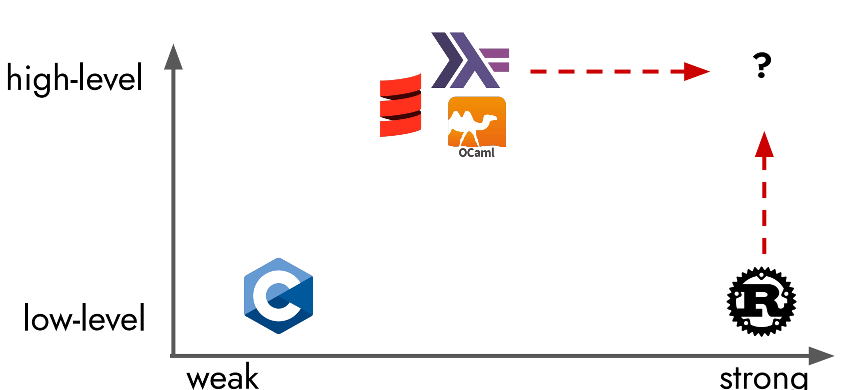
Motivation

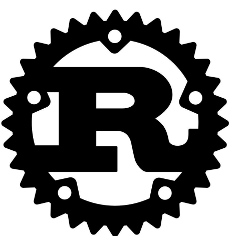




What would post-Rust languages resemble?

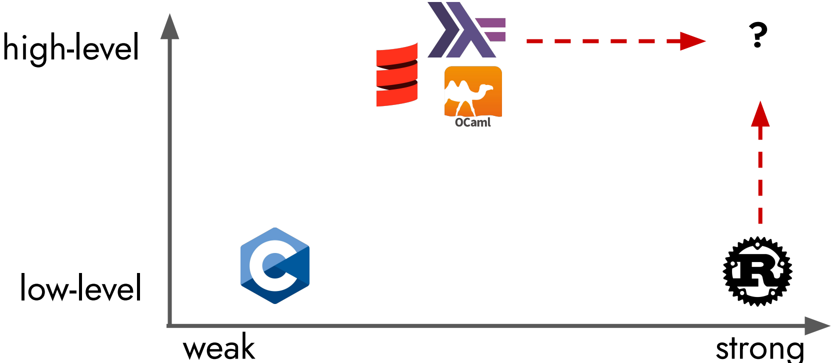
Motivation





What would post-Rust languages resemble?

Motivation



Ownership and Lifetimes

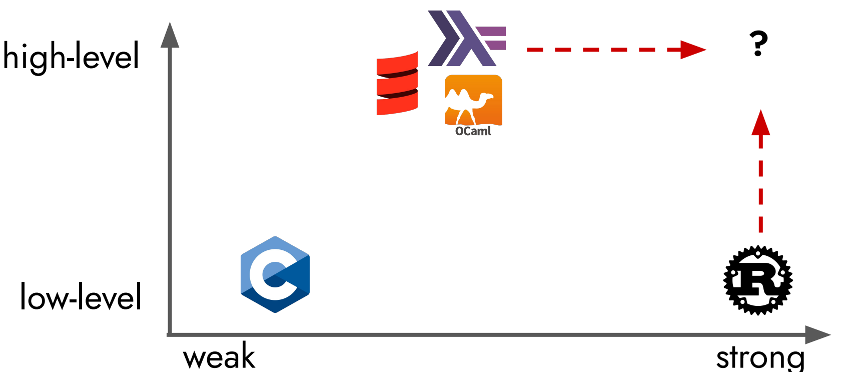
The ownership system is partially implemented, and is expected to get built out in the next couple of months. The basic support for ownership includes features like:

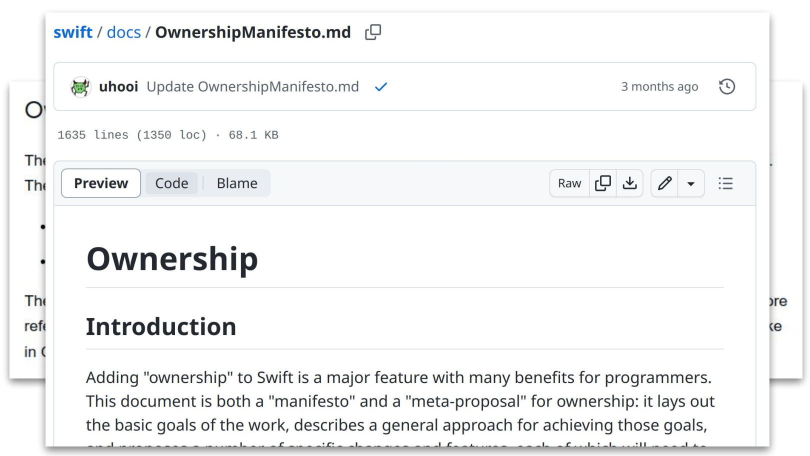
- Capture declarations in closures.
- Borrow checker: complain about invalid mutable references.

The next step in this is to bring proper lifetime support in. This will add the ability to return references and store references in structures safely. In the immediate future, one can use the unsafe `Pointer` struct to do this like in C++.

Mojo / Modular.ai

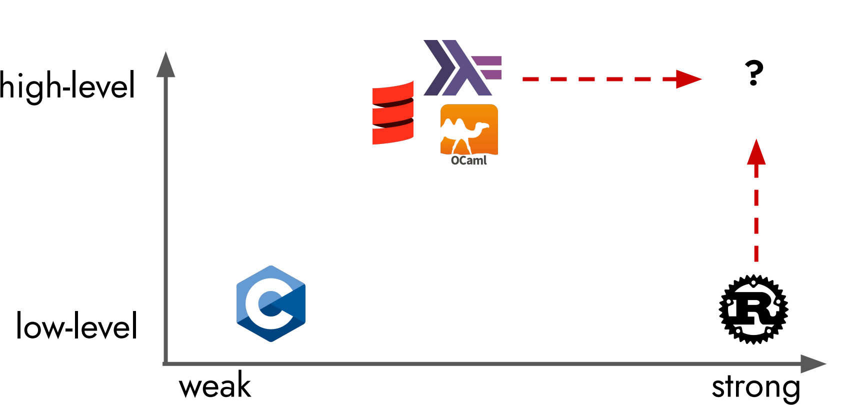
Motivation





Swift / Apple

Motivation



Linear Haskell

Practical Linearity in a Higher-Order Polymorphic Language

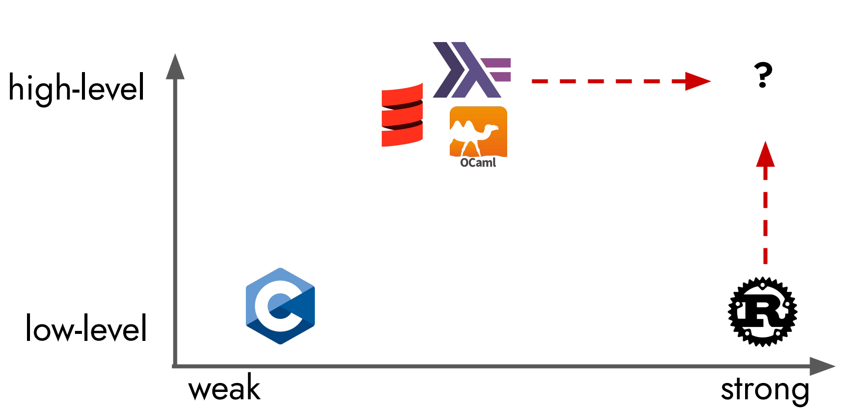
JEAN-PHILIPPE BERNARDY, University of Gothenburg, Sweden
MATHIEU BOESPFLUG, Tweag I/O, France
RYAN R. NEWTON, Indiana University, USA
SIMON PEYTON JONES, Microsoft Research, UK
ARNAUD SPIWACK, Tweag I/O, France

Linear type systems have a long and storied history, but not a clear path forward to integrate with existing languages such as OCaml or Haskell. In this paper, we study a linear type system designed with two crucial properties in mind: backwards-compatibility and code reuse across linear and non-linear users of a library. Only then can the benefits of linear types permeate conventional functional programming. Rather than bifurcate types into linear and non-linear counterparts, we instead attach linearity to *function arrows*. Linear functions can receive inputs from linearly-bound values, but can *also* operate over unrestricted, regular values.

To demonstrate the efficacy of our linear type system — both how easy it can be integrated in an existing language implementation and how streamlined it makes it to write programs with linear types — we implemented our type system in GHC, the leading Haskell compiler, and demonstrate two kinds of applications of linear types: mutable data with pure interfaces; and enforcing protocols in I/O-performing functions.

Substructural type systems
e.g. Linear Haskell [POPL 2018]

Motivation



Capturing Types

ALEKSANDER BORUCH-GRUSZECKI and MARTIN ODERSKY, EPFL
EDWARD LEE and ONDŘEJ LHOTÁK, University of Waterloo
JONATHAN BRACHTHÄUSER, Eberhard Karls University of Tübingen

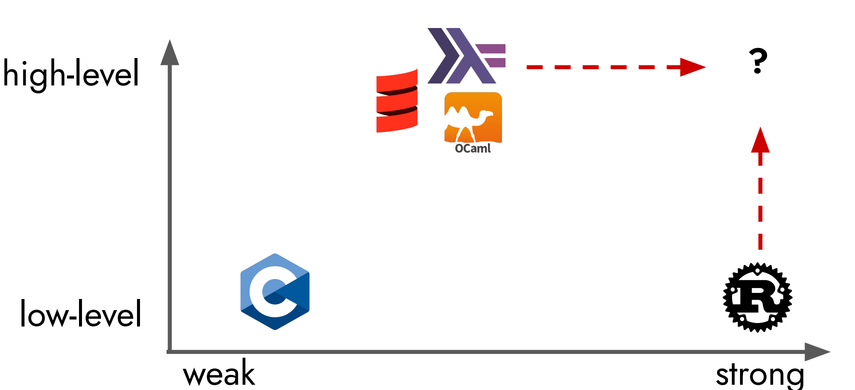
Type systems usually characterize the shape of values but not their free variables. However, many desirable safety properties could be guaranteed if one knew the free variables captured by values. We describe $CC_{<:\square}$, a calculus where such captured variables are succinctly represented in types, and show it can be used to safely implement effects and effect polymorphism via scoped capabilities. We discuss how the decision to track captured variables guides key aspects of the calculus, and show that $CC_{<:\square}$ admits simple and intuitive types for common data structures and their typical usage patterns. We demonstrate how these ideas can be used to guide the implementation of capture checking in a practical programming language.

CCS Concepts: • **Theory of computation** → **Type structures**; • **Software and its engineering** → *Object oriented languages;*

Additional Key Words and Phrases: Scala, type systems, effects, resources, capabilities

Scala 3 Capturing Types [TOPLAS 2023]

Motivation



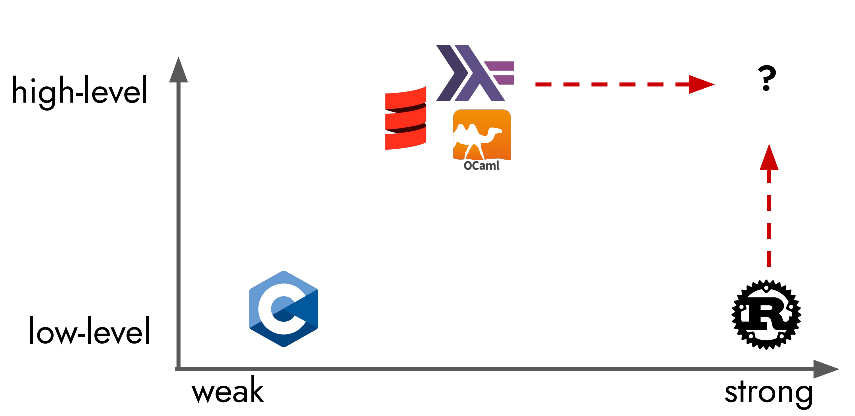
Data Race Freedom à la Mode

AÏNA LINN GEORGES, MPI-SWS, Germany
BENJAMIN PETERS, MPI-SWS, Germany
LAILA ELBEHEIRY, MPI-SWS, Germany
LEO WHITE, Jane Street, UK
STEPHEN DOLAN, Jane Street, UK
RICHARD A. EISENBERG, Jane Street, USA
CHRIS CASINGHINO, Jane Street, USA
FRANÇOIS POTTIER, Inria, France
DEREK DREYER, MPI-SWS, Germany

We present DRFCaml, an extension of OCaml's type system that guarantees data race freedom for multi-threaded OCaml programs while retaining backward compatibility with existing sequential OCaml code. We build on recent work of Lorenzen et al., who extend OCaml with *modes* that keep track of locality, uniqueness, and affinity. We introduce two new mode axes, *contention* and *portability*, which record whether data has been shared or can be shared between multiple threads. Although this basic type-and-mode system has limited expressive power by itself, it does let us express APIs for *capsules*, regions of memory whose access is controlled by a unique ghost key, and *reader-writer locks*, which allow a thread to safely acquire partial or full ownership of a key. We show that this allows complex data structures (which may involve aliasing and mutable state) to be safely shared between threads. We formalize the complete system and establish its soundness by building a semantic model of it in the Iris program logic on top of the Rocq proof assistant.

OCaml with Rust-style Ownership [POPL 25]

Motivation



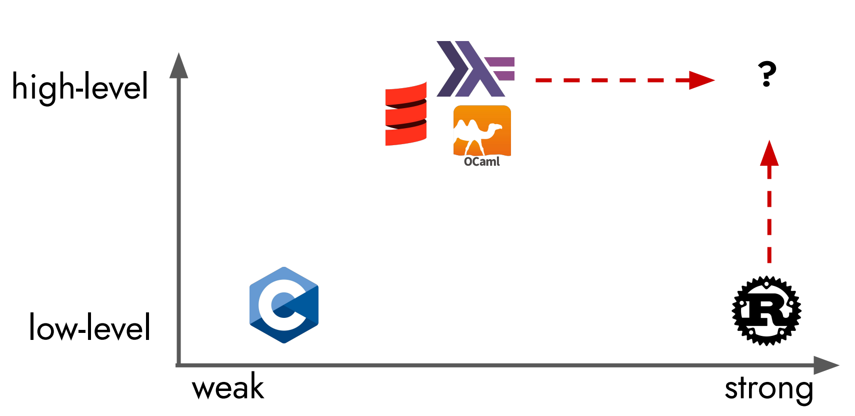
How to smoothly combine functional/type abstractions with resource tracking/control?

Rust's "*secret sauce*"

vs

Pervasive sharing from functional abstraction

Motivation



How to smoothly combine functional/type abstractions with resource tracking/control?

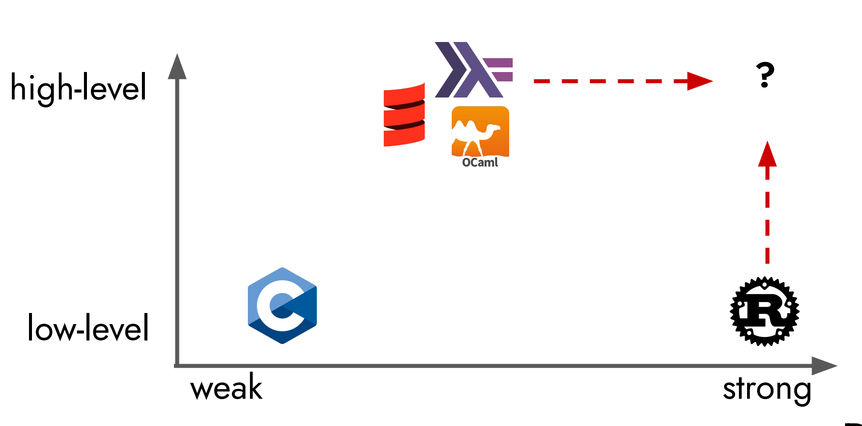
"shared XOR mutable"

Rust's ~~*"secret sauce"*~~

vs

Pervasive sharing from functional abstraction

Motivation



How to smoothly combine functional/type abstractions with resource tracking/control?

"shared XOR mutable"

Rust's ~~"secret sauce"~~

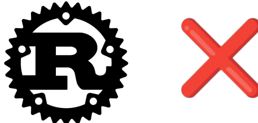
vs

Pervasive **sharing** from functional abstraction

first class functions, capturing, escaping ...

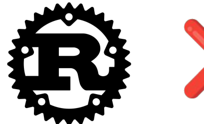
Example: Pair of Counters

```
val c = new Ref(n)
(() => c += 1, () => c -= 1)
```



Example: Pair of Counters

```
def counter(n: Int): Pair[() => Unit, () => Unit] = {
  val c = new Ref(n)
  (() => c += 1, () => c -= 1)
}
```

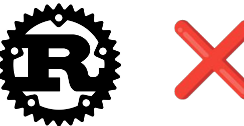


❌

Example: Pair of Counters

```
def counter(n: Int): Pair[() => Unit, () => Unit] = {
  val c = new Ref(n)
  (() => c += 1, () => c -= 1)
}

val ctr = counter(0)
// ctr: Pair[(() => Unit)  , (() => Unit)  ]
val incr = fst(ctr)  // incr: (() => Unit)
val decr = snd(ctr)  // decr: (() => Unit)
```

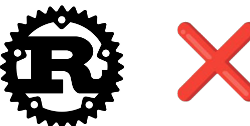


Example: Pair of Counters

```
def counter(n: Int): Pair[() => Unit, () => Unit] = {
  val c = new Ref(n)
  (() => c += 1, () => c -= 1)
}

val ctr = counter(0)
// ctr: Pair[(() => Unit)  , (() => Unit)  ]
val incr = fst(ctr)  // incr: (() => Unit)
val decr = snd(ctr)  // decr: (() => Unit)

par { () => incr() } { () => decr() }   // UNSAFE!
```

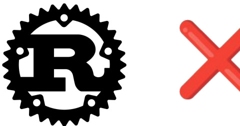
 ❌

Example: Pair of Counters

```
def counter(n: Int): Pair[() => Unit, () => Unit] = {
  val c = new Ref(n)
  (() => c += 1, () => c -= 1)
}

val ctr = counter(0)
// ctr: Pair[(() => Unit)  , (() => Unit)  ]
val incr = fst(ctr)  // incr: (() => Unit)
val decr = snd(ctr)  // decr: (() => Unit)

par { () => incr() } { () => decr() }    // UNSAFE!
incr(); decr()                           // SAFE!
```



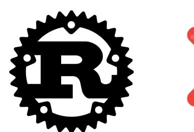
❌

Example: Pair of Counters

```
def counter(n: Int): Pair[() => Unit, () => Unit] = {
  val c = new Ref(n)
  (() => c += 1, () => c -= 1)
}

val ctr = counter(0)
// ctr: Pair[(() => Unit)^ctr, (() => Unit)^ctr]^ctr
val incr = fst(ctr)  // incr: (() => Unit)^ctr
val decr = snd(ctr)  // decr: (() => Unit)^ctr

par { () => incr() } { () => decr() }    // UNSAFE!
incr(); decr()                           // SAFE!
```



❌

Reachability Types

Reachability Types

Rust, state-of-the-art ownership type systems

Reachability types, separation logic,

Borrowing: temporarily relax access where needed

Ownership: unique access paths, global heap invariant

Uniqueness, separation: restrict access where needed

Sharing, reachability: flexible heap properties, no globally enforced invariants

Strict foundation, selectively relaxed.

Liberal foundation, selectively restricted.

Reachability Types: tracking sharing and separation in higher-order languages
[OOPSLA 2021, Bao et al.]

Polymorphic Reachability Types

Rust, state-of-the-art ownership type systems

Borrowing: temporarily relax access where needed
Ownership: unique access paths, global heap invariant

Strict foundation, selectively relaxed.

Reachability types, separation logic,

Uniqueness, separation: restrict access where needed
Sharing, reachability: flexible heap properties, no globally enforced invariants

Liberal foundation, selectively restricted.

Reachability Types: tracking sharing and separation in higher-order languages
[OOPSLA 2021, Bao et al.]

This work [POPL 2024]
A new formulation of reachability types

- ★ smoothly combine with polymorphism
- ★ a notion of contextual freshness
- ★ precise lightweight reachability polymorphism
- ★ bounded type-and-reachability polymorphism

Polymorphic Reachability Types – Agenda

1. **Basic reachability tracking mechanism**
2. Formalization and metatheory
3. Effect system extension
4. Applications
5. Conclusion and future work

Qualifying Types with a Set of Variables

- **Key idea**: $\Gamma \vdash e : T^{q}$

  q the set of variables that can be reached from the evaluation result of e.

- ```
  val x = new Ref(42) // x : Ref[Int]^x
  val y = x           // y : Ref[Int]^y in context [ y: Ref[Int]^x, ... ]
  val i = 42          // i : Int^∅, untracked
  ```
  ```

Qualifying Types with a Set of Variables

- **Key idea**: $\Gamma \vdash e : T^{q}$

  q the set of variables that can be reached from the evaluation result of e.

- ```
  val x = new Ref(42) // x : Ref[Int]^x
  val y = x           // y : Ref[Int]^y in context [ y: Ref[Int]^x, ... ]
  val i = 42          // i : Int^∅, untracked
  ```

- Function types track the observable context:

  ```
  val c = new Ref(42)
  (n: Int) => { c := n } // : (Int => Unit)^{c}
  ```
  ```

Qualifying Types with ... Not Yet a Variable?

- Key idea: $\Gamma \vdash e : T^{q}$

  q the set of variables that can be reached from the evaluation result of e.

- **What should be the qualifier for fresh allocations?**

```
new Ref(42)            // : Ref[Int]?
```

Qualifying Types with ... Not Yet a Variable?

- Key idea: $\Gamma \vdash e : T^{q}$

  q the set of variables that can be reached from the evaluation result of e.

- **What should be the qualifier for fresh allocations?**

```
new Ref(42)              // : Ref[Int]^⊥
```

Possible option 1: $\bot$ shared nothing, but confused with untracked!

Either unsound if not distinguished from untracked (primitive) values,
or the system becomes non-parametric over the untracked (as in Bao et al.).

Qualifying Types with ... Not Yet a Variable?

- Key idea: $\Gamma \vdash e : T^{q}$

  q the set of variables that can be reached from the evaluation result of e.

- **What should be the qualifier for fresh allocations?**

  ```
  new Ref(42)            // : Ref[Int]^⊤
  ```

  Possible option 2: $\top$ can be potentially shared with everything, but not really!

  I.e. the universal/root capability `{cap}` in Scala Capturing Types,
  where additional mechanisms are required to establish separation.

A New Notion of Freshness

- **Key idea**: use a special marker ◆ to represent statically unobservable variables/locations.

```
new Ref(42)                 // : Ref[Int]◆, fresh allocation
```

A New Notion of Freshness

- **Key idea**: use a special marker ◆ to represent statically unobservable variables/locations.

```
new Ref(42)             // : Ref[Int]◆, fresh allocation
```

Unobservable variables/locations may materialize during evaluation:

```
new Ref(42) → ℓ         // : Ref[Int]{ℓ}
```

A New Notion of Freshness

- **Key idea**: use a special marker ◆ to represent statically unobservable variables/locations.

```
new Ref(42)            // : Ref[Int]◆, fresh allocation
```

Unobservable variables/locations may materialize during evaluation:

```
new Ref(42) → ℓ        // : Ref[Int]{ℓ}
```

Bound/known reachability sets cannot upcast to ◆:

```
val x = new Ref(42)    // : Ref[Int]x not subtype of Ref[Int]◆
```

A New Notion of Freshness

- **Key idea**: use a special marker ◆ to represent statically unobservable variables/locations.

  ```
  new Ref(42)              // : Ref[Int]◆, fresh allocation
  ```

  Unobservable variables/locations may materialize during evaluation:

  ```
  new Ref(42) → ℓ          // : Ref[Int]{ℓ}
  ```

  Bound/known reachability sets cannot upcast to ◆:

  ```
  val x = new Ref(42)      // : Ref[Int]x not subtype of Ref[Int]◆
  ```

- Leads to a parametric treatment of reachability/separation;
  No conflation of *untracked* vs *fresh* resources anymore (cf. Bao et al.).

A New Notion of Freshness

- **Key idea**: use a special marker ◆ to represent statically unobservable variables/locations.

- Support both scoped and non-scoped introduction forms of resources:

```
def try[A](f: CanThrow◆ => A): A
try { throw => ... }
```

A New Notion of Freshness

- **Key idea**: use a special marker ◆ to represent statically unobservable variables/locations.

- Support both scoped and non-scoped introduction forms of resources:

```
def try[A](f: CanThrow◆ => A): A
try { throw => ... }

try[CanThrow◆] { throw => throw } // error:
                                  // CanThrowᶜ not subtype of CanThrow◆
```

A New Notion of Freshness

- **Key idea**: use a special marker ◆ to represent statically unobservable variables/locations.

- Support both scoped and non-scoped introduction forms of resources:

```
def try[A](f: CanThrow◆ => A): A
try { throw => ... }

try[CanThrow◆] { throw => throw } // error:
                                  // CanThrowᶜ not subtype of CanThrow◆
try[Ref[Int]◆] { throw => new Ref(42) } // okay
```

A New Notion of Freshness

- **Key idea**: use a special marker ◆ to represent statically unobservable variables/locations.

- Support both scoped and non-scoped introduction forms of resources:

```
def try[A](f: CanThrow◆ => A): A
try { throw => ... }

try[CanThrow◆] { throw => throw } // error:
                                  // CanThrowᶜ not subtype of CanThrow◆
try[Ref[Int]◆] { throw => new Ref(42) } // okay
```

- More unified treatment compared with Scala Capturing Types:

```
try[Ref[Int]ᵀ](c => new Ref(42)) // has to diff. {cap} and {ref}
```

From Freshness to Separation

- In intersection type systems

```
Int & String <: Nothing // not typically derivable in syntactic subtyping
```

From Freshness to Separation

- In intersection type systems

```
Int & String <: Nothing // not typically derivable in syntactic subtyping
```

- Our freshness marker

$\forall$ x *in the typing context,* `locs(◆) ∩ locs(x) ⊆ ∅`

From Freshness to Separation

- In intersection type systems
  `Int & String <: Nothing // not typically derivable in syntactic subtyping`
- Our freshness marker
  ∀ x *in the typing context*, $\mathtt{locs}(◆) \cap \mathtt{locs}(x) \subseteq \varnothing$
- **Key Idea**: *observable separation* between arguments and the function
  ∀ x *in function observation*, $\mathtt{locs}(◆) \cap \mathtt{locs}(x) \subseteq \varnothing$

From Freshness to Separation

- In intersection type systems
  `Int & String <: Nothing // not typically derivable in syntactic subtyping`
- Our freshness marker
  ∀x *in the typing context*, $\texttt{locs}(\blacklozenge) \cap \texttt{locs}(x) \subseteq \varnothing$
- **Key Idea**: *observable separation* between arguments and the function
  ∀x *in function observation*, $\texttt{locs}(\blacklozenge) \cap \texttt{locs}(x) \subseteq \varnothing$

```
def id(x: T◆): T{x} = x // : ((x: T◆) => T{x})∅
id(y)                   // okay
id(new Ref(42))         // okay
```

Checking Separation

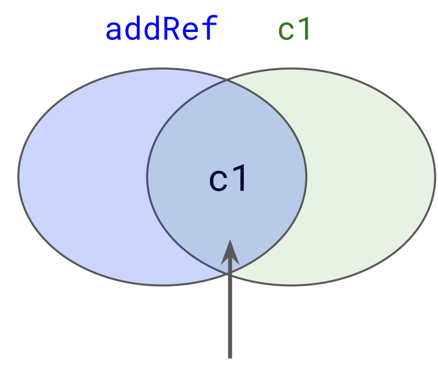
- Applications check separation by non-overlapping reachability:

```
val c1: Ref[Int]^{c1}
val c2: Ref[Int]^{c2}
def addRef(r: Ref[Int]◆) = { c1 := !c1 + !r; c1 }
```

Checking Separation

- Applications check separation by non-overlapping reachability:

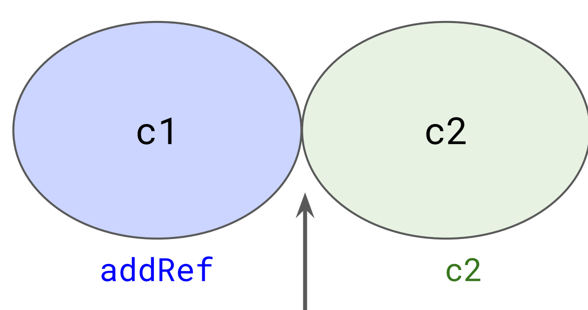
```
val c1: Ref[Int]^{c1}
val c2: Ref[Int]^{c2}
def addRef(r: Ref[Int]◆) = { c1 := !c1 + !r; c1 }
addRef(c1) // type error because {c1} ∩ {c1} ⊈ ∅
```



Checking Separation

- Applications check separation by non-overlapping reachability:

```
val c1: Ref[Int]^{c1}
val c2: Ref[Int]^{c2}
def addRef(r: Ref[Int]^◆) = { c1 := !c1 + !r; c1 }
addRef(c1) // type error because {c1} ∩ {c1} ⊈ ∅
addRef(c2) // ok because {c2} ∩ {c1} ⊆ ∅
```



Checking Separation

- Applications check separation by non-overlapping reachability:

```
val c1: Ref[Int]^{c1}
val c2: Ref[Int]^{c2}
def addRef(r: Ref[Int]^◆) = { c1 := !c1 + !r; c1 }
addRef(c1) // type error because {c1} ∩ {c1} ⊈ ∅
addRef(c2) // ok because {c2} ∩ {c1} ⊆ ∅
```

Invariant: non-overlapping *reachability* implies separate *locations*!

Checking Separation

- Applications check separation by non-overlapping reachability:

```
val c1: Ref[Int]^{c1}
val c2: Ref[Int]^{c2}
def addRef(r: Ref[Int]^◆) = { c1 := !c1 + !r; c1 }
addRef(c1) // type error because {c1} ∩ {c1} ⊈ ∅
addRef(c2) // ok because {c2} ∩ {c1} ⊆ ∅
```

Invariant: non-overlapping *reachability* implies separate *locations*!

- Function argument qualifier describes permissible overlap/aliasing patten:

```
def addRef2(c: Ref[Int]^{c1, ◆}) = …
addRef2(c1) // ok now {c1} ∩ {c1} ⊆ {c1}
```

Lightweight Reachability Polymorphism

- Lightweight (quantification-free) reachability polymorphism:

```
def id[T](x: T◆): T{x} = x  // : ((x: T◆) => T{x})∅
id(42)                     // : Int∅
id(new Ref(42))            // : Ref[Int]◆
id(x)                      // : Ref[Int]{x}
```

Term-level variables already indicate the dependency: result reachability can precisely depend on the argument reachability without quantification.

Bounded Reachability Polymorphism

- Bounded parametric reachability a la $F_{<:}$

```
def id[Tᶻ <: Top◆](x: Tᶻ): T{z} = x
def fst[Aᵃ <: Top◆, Bᵇ <: Top◆](p: Pair[Aᵃ,Bᵇ]): Aᵃ = ...

val v1 = new Ref(1)
val v2 = new Ref(2)
val p = makePair(v1, v2) // : Pair[Ref[Int]^v1, Ref[Int]^v2]
fst(p)                   // : Ref[Int]^v1
```

Polymorphic Reachability Types – Agenda

1. Basic reachability tracking mechanism
2. **Formalization and metatheory**
3. Effect system extension
4. Applications
5. Conclusion and future work

Formalization

$$\Gamma^{\varphi} \vdash e : T^{q}$$

***Context**: what resources can be observed*

- lexical scope, capturing, escaping

***Space**: where are things/heap topology*

- reachability, aliasing/sharing, separation

Term Typing $\boxed{\Gamma^{\varphi} \vdash t : Q}$

$$\frac{x : T^{q} \in \Gamma \qquad x \in \varphi}{\Gamma^{\varphi} \vdash x : T^{x}} \quad \text{(T-VAR)}$$

$$\frac{c \in B}{\Gamma^{\varphi} \vdash c : B^{\varnothing}} \quad \text{(T-CST)}$$

$$\frac{(\Gamma,\ f : F,\ x : P)^{q,x,f} \vdash t : Q \qquad q \subseteq \varphi \qquad F = (f(x : P) \to Q)^{q}}{\Gamma^{\varphi} \vdash \lambda f(x).t : F} \quad \text{(T-ABS)}$$

$$\frac{\Gamma^{\varphi} \vdash t : T^{q} \qquad \blacklozenge \notin q}{\Gamma^{\varphi} \vdash \mathsf{ref}\ t : (\mathsf{Ref}\ T^{q})^{\blacklozenge q}} \quad \text{(T-REF)}$$

$$\frac{\Gamma^{\varphi} \vdash t : (\mathsf{Ref}\ T^{p})^{q} \qquad \blacklozenge \notin p \qquad p \subseteq \varphi}{\Gamma^{\varphi} \vdash !t : T^{p}} \quad \text{(T-DEREF)}$$

$$\frac{\Gamma^{\varphi} \vdash t_1 : (f(x : T^{p}) \to Q)^{q} \qquad \Gamma^{\varphi} \vdash t_2 : T^{p} \qquad \blacklozenge \notin p \qquad Q = U^{r} \qquad r \subseteq \blacklozenge\varphi, x, f \qquad f \notin \mathrm{fv}(U)}{\Gamma^{\varphi} \vdash t_1\ t_2 : Q[p/x, q/f]} \quad \text{(T-APP)}$$

$$\frac{\Gamma^{\varphi} \vdash t_1 : (\mathsf{Ref}\ T^{p})^{q} \qquad \Gamma^{\varphi} \vdash t_2 : T^{p} \qquad \blacklozenge \notin p}{\Gamma^{\varphi} \vdash t_1 := t_2 : \mathsf{Unit}^{\varnothing}} \quad \text{(T-ASSGN)}$$

$$\frac{\Gamma^{\varphi} \vdash t_1 : (f(x : T^{p \cap q}) \to Q)^{q} \qquad \Gamma^{\varphi} \vdash t_2 : T^{p} \qquad Q = U^{r} \qquad r \subseteq \blacklozenge\varphi, x, f \qquad \blacklozenge \in p \Rightarrow x \notin \mathrm{fv}(U) \qquad f \notin \mathrm{fv}(U)}{\Gamma^{\varphi} \vdash t_1\ t_2 : Q[p/x, q/f]} \quad \text{(T-APP}\blacklozenge\text{)}$$

$$\frac{\Gamma^{\varphi} \vdash t : Q \qquad \Gamma \vdash Q <: T^{q} \qquad q \subseteq \blacklozenge\varphi}{\Gamma^{\varphi} \vdash t : T^{q}} \quad \text{(T-SUB)}$$

- Simply-typed $\lambda^{\blacklozenge}$-calculus
- $F^{\blacklozenge}_{<:}$-calculus with bounded polymorphism

More details in Wei et al. POPL 2024

Formalization

$$\Gamma^{\varphi} \vdash e : T^{q}$$

***Context**: what resources can be observed*

- lexical scope, capturing, escaping

***Space**: where are things/heap topology*

- reachability, aliasing/sharing, separation

$$\frac{x : T^{q} \in \Gamma \qquad x \in \varphi}{\Gamma^{\varphi} \vdash x : T^{x}} \quad \text{(T-VAR)}$$

- Simply-typed $\lambda^{\blacklozenge}$-calculus
- $F^{\blacklozenge}_{<:}$-calculus with bounded polymorphism

Formalization

$$\Gamma^{\varphi} \vdash e : T^{q}$$

***Context**: what resources can be observed*

- lexical scope, capturing, escaping

***Space**: where are things/heap topology*

- reachability, aliasing/sharing, separation

$$\frac{(\Gamma, f : F, x : P)^{q,x,f} \vdash t : Q \quad q \subseteq \varphi \qquad F = (f(x : P) \to Q)^{q}}{\Gamma^{\varphi} \vdash \lambda f(x).t : F} \text{(T-ABS)}$$

- Simply-typed $\lambda^{\blacklozenge}$-calculus
- $F^{\blacklozenge}_{<:}$-calculus with bounded polymorphism

Formalization

$$\Gamma^{\varphi} \vdash e : T^{q}$$

***Context**: what resources can be observed*

- lexical scope, capturing, escaping

***Space**: where are things/heap topology*

- reachability, aliasing/sharing, separation

$$\frac{\Gamma^{\varphi} \vdash t_1 : (f(x : T^{p \cap q}) \to Q)^{q} \quad \Gamma^{\varphi} \vdash t_2 : T^{p} \quad Q = U^{r} \quad r \subseteq \blacklozenge\varphi, x, f \quad \blacklozenge \in p \Rightarrow x \notin \mathrm{fv}(U) \quad f \notin \mathrm{fv}(U)}{\Gamma^{\varphi} \vdash t_1\, t_2 : Q[p/x, q/f]} \; (\text{T-APP}\blacklozenge)$$

- Simply-typed $\lambda^{\blacklozenge}$-calculus
- $F^{\blacklozenge}_{<:}$-calculus with bounded polymorphism

Formalization

$$\Gamma^{\varphi} \vdash e : T^{q}$$

***Context**: what resources can be observed*

- lexical scope, capturing, escaping

***Space**: where are things/heap topology*

- reachability, aliasing/sharing, separation

- Simply-typed $\lambda^{\blacklozenge}$-calculus
- $F^{\blacklozenge}_{<:}$-calculus with bounded polymorphism

$$\frac{\Gamma^{\varphi} \vdash t : T^{q} \qquad \blacklozenge \notin q}{\Gamma^{\varphi} \vdash \mathsf{ref}\ t : (\mathsf{Ref}\ T^{q})^{\blacklozenge q}} \quad (\text{T-REF})$$

Unsound:

```
val x = new Ref(new Ref(42))
!x // : Ref[Int]◆
!x // : Ref[Int]◆
```

Formalization

$$\Gamma^{\varphi} \vdash e : T^{q}$$

***Context**: what resources can be observed*

- lexical scope, capturing, escaping

***Space**: where are things/heap topology*

- reachability, aliasing/sharing, separation

- Simply-typed $\lambda^{\blacklozenge}$-calculus
- $F^{\blacklozenge}_{<:}$-calculus with bounded polymorphism

$$\frac{\Gamma^{\varphi} \vdash t : T^{q} \qquad \blacklozenge \notin q}{\Gamma^{\varphi} \vdash \mathsf{ref}\ t : (\mathsf{Ref}\ T^{q})^{\blacklozenge q}} \quad \text{(T-REF)}$$

Unsound:

```
val x = new Ref(new Ref(42))
!x // : Ref[Int]◆
!x // : Ref[Int]◆
```

With freshness restriction:

```
val r = new Ref(42)
val x = new Ref(r)
!x // : Ref[Int]ʳ
!x // : Ref[Int]ʳ
```

Metatheory

- Syntactic soundness
  - Progress
  - Preservation: qualifiers may grow only due to freshness (new allocations)
- Preservation of Separation: two separate terms remain separate after reduction steps.

Metatheory

- Syntactic soundness
  - Progress
  - Preservation: qualifiers may grow only due to freshness (new allocations)
- Preservation of Separation: two separate terms remain separate after reduction steps.

THEOREM 4.7 (PRESERVATION). *If* $[\varnothing \mid \Sigma]^{\varphi} \vdash t : T^{q}$, *and* $[\varnothing \mid \Sigma]^{\varphi} \vdash \sigma$, *and* $t \mid \sigma \to t' \mid \sigma'$, *and* Σ ok, *then there exists* $\Sigma' \supseteq \Sigma$, $\varphi' \supseteq \varphi \cup p$, *and* $p \subseteq \mathrm{dom}(\Sigma' \setminus \Sigma)$ *such that* $[\varnothing \mid \Sigma']^{\varphi'} \vdash \sigma'$ *and* $[\varnothing \mid \Sigma']^{\varphi'} \vdash t' : T^{q[p/\blacklozenge]}$.

Metatheory

- Syntactic soundness
  - Progress
  - Preservation: qualifiers may grow only due to freshness (new allocations)
- Preservation of Separation: two separate terms remain separate after reduction steps.

Corollary 4.8 (Preservation of Separation). *Sequential reduction of two terms with disjoint qualifiers preserve types and disjointness:*

$$\frac{\begin{array}{llll}[\varnothing \mid \Sigma]^{\mathrm{dom}(\Sigma)} \vdash t_1 : T_1^{q_1} & t_1 \mid \sigma \to t_1' \mid \sigma' & \varnothing \mid \Sigma \vdash \sigma & \Sigma \text{ ok} \\ [\varnothing \mid \Sigma]^{\mathrm{dom}(\Sigma)} \vdash t_2 : T_2^{q_2} & t_2 \mid \sigma' \to t_2' \mid \sigma'' & q_1 \cap q_2 \subseteq \{\blacklozenge\} & \end{array}}{\begin{array}{lll}\exists p_1\ p_2\ \Sigma'\ \Sigma''. & [\varnothing \mid \Sigma'\ ]^{\mathrm{dom}(\Sigma')} \vdash t_1' : T_1^{p_1} & \Sigma'' \supseteq \Sigma' \supseteq \Sigma \\ & [\varnothing \mid \Sigma'']^{\mathrm{dom}(\Sigma'')} \vdash t_2' : T_2^{p_2} & p_1 \cap p_2 \subseteq \{\blacklozenge\}\end{array}}$$

Mechanization & Implementation

- Mechanized formalization in Rocq/Coq
  - Alternative logical relation formalization in progress
- Prototype implementation Diamond language
- Both can be found at *https://github.com/TiarkRompf/reachability*

Reachability Types

Reachability types are a new take on modeling lifetimes and sharing in high-level functional languages, showing how to integrate Rust-style reasoning capabilities with higher-order functions, polymorphic types, and similar high-level abstractions.

Mechanization Overview

- `base` -- Coq mechanization of the λ^*-calculus [1] and its variations, gradually increasing in complexity.
- `effects` -- Coq mechanization of the λ^*_ε-calculus [1] and its variations, gradually increasing in complexity.
- `polymorphism` -- Coq mechanization of the $\lambda^\diamond$-calculus [2] and its variations, featuring a refined reachability model that scales to parametric type polymorphism.
- `log-rel-unary` -- Unary logical relations for proving semantic type soundness and termination of $\lambda^\diamond$, $\lambda^\diamond_\varepsilon$, and its variants [4,5].
- `log-rel-binary` -- Binary logical relations for establishing equational reasoning about $\lambda^\diamond$, $\lambda^\diamond_\varepsilon$, and its variants [4].
- `log-rel-step-indexed` -- Step-indexed logical relations for $\lambda^\diamond$, $\lambda^\diamond_\varepsilon$ and its variants [4].
- `checking` -- Bidirectional type system $\lambda^\diamond_R$ with decidable type checking/inference, including refined subtyping for self-references [5]

Polymorphic Reachability Types – Agenda

1. Basic reachability tracking mechanism
2. Formalization and metatheory
3. **Effect system extension**
4. Applications
5. Conclusion and future work

Flow-Sensitive Effect System

- **Key idea**: Tracking side-effects with aliases, separation, and flow-sensitivity

$$\Gamma^{\varphi} \vdash e : T^{q} \mid \varepsilon$$

An alias-aware, flow-sensitive effect algebra

```
val x = new Ref(0) // : Ref[Int]^x
val y = id(x)      // : Ref[Int]^{y,x}
y := 42            // : Unit @wr(y,x)
```

Flow-Sensitive Effect System

- **Key idea**: Tracking side-effects with aliases, separation, and flow-sensitivity

$$\Gamma^{\varphi} \vdash e : T^{q} \mid \varepsilon$$

An alias-aware, flow-sensitive effect algebra

- Effect quantale [Gordon 2021, TOPLAS]
  An effect quantale Q = (E, ⊔, ▸, I) is a partial (binary) join semilattice (E, ⊔) with partial monoid (E, ▸, I)
  - ⊔ models commutative effect join (e.g. two branches)
  - ▸ models sequential effect composition

Lifting to Store-Sensitive Effect System

- Effect store `ε ∈ P(P(Var) x PreEff)` is an instance of effect quantale if `PreEff` is an effect quantale too.

  Example: { {x, y} → Read, {w, v} → Write } where PreEff = {Read, Write}

Lifting to Store-Sensitive Effect System

- Effect store $\varepsilon \in P(P(Var) \times PreEff)$ is an instance of effect quantale if $PreEff$ is an effect quantale too.

  Example: $\{ \{x, y\} \rightarrow Read, \{w, v\} \rightarrow Write \}$ where $PreEff = \{Read, Write\}$

$$\{ \{x, y\} \rightarrow Read \} \sqcup \{ \{y, z\} \rightarrow Write \}$$

$$= \{ \{x, y\} \cup \{y, z\} \rightarrow Read \sqcup_{PreEff} Write \} \quad \text{since } \{x, y\} \cap \{y, z\} \neq \emptyset$$

$$= \{ \{x, y, z\} \rightarrow Write \}$$

Destructive Effects

- **Key idea**: ▸ doesn't have to be commutative, we define a "kill" effect that can only be partially composed (see details in Bao et al., OOPSLA 21)

```
read ▸ kill // okay
kill ▸ read // error
```

Destructive Effects

- **Key idea**: ▸ doesn't have to be commutative, we define a "kill" effect that can only be partially composed (see details in Bao et al., OOPSLA 21)

```
read ▸ kill // okay
kill ▸ read // error
```

```
var c1 = ...
val x = move(c1) // : Ref[Int]◆ @kill(c1)
c1 += 1          // error
```

$\triangleright$	$\bot_{\mathbb{E}}$	rd	wr	kill
$\bot_{\mathbb{E}}$	$\bot_{\mathbb{E}}$	rd	wr	kill
rd	rd	rd	wr	kill
wr	wr	wr	wr	kill
kill	undefined			

Polymorphic Reachability Types – Agenda

1. Basic reachability tracking mechanism
2. Formalization and metatheory
3. Effect system extension
4. **Applications**
5. Conclusion and future work

Applications

- **Flexible and safe programming with resources** [POPL 24, OOPSLA 21]
  - Safe parallelization
  - Scoped capabilities
  - Ownership transfer and affinity
  - ...

- Optimizations for high-order effectful programs [OOPSLA 23]
  - Effect-guided optimization
    - Dead-write elimination
    - Constant sub-expression elimination
  - Code motion (lambda-hoisting)
  - ...

Checking Separation -- Safe Parallelization

- Requiring disjoint qualifiers of two thunks to ensure non-interference:

```
// library code
def par(a: (() => Unit)◆)(b: (() => Unit)◆): Unit

// user code
val c1 = new Ref(0), c2 = new Ref(0)
par {
  // ok: operate on c1 only, cannot access c2
  c1 += 42
} {
  // ok: operate on c2 only, cannot access c1
  c2 -= 100
}
```

Non-Escaping Scoped Capabilities

```
val res = withFile("a.txt") { file =>
  val line = file.readLine()
  ...
}
```

`withFile` abstraction ensures the file is closed after use.

More details/examples in OOPSLA 21, POPL 24.

Non-Escaping Scoped Capabilities

```
val res = withFile("a.txt") { file =>
  val line = file.readLine()
  ...
  { () => file.readLine() }
}

val f = res()
```

More details/examples in OOPSLA 21, POPL 24.

Non-Escaping Scoped Capabilities

```
val res = withFile("a.txt") { file =>
  val line = file.readLine()
  ...
  { () => file.readLine() }
}

val f = res()
```

file handle escaped!

More details/examples in OOPSLA 21, POPL 24.

Non-Escaping Scoped Capabilities

```
val res = withFile("a.txt") { file =>
  val line = file.readLine()
  ...
  { () => file.readLine() }
}
```

file handle escaped!

```
val f = res()
```

at this moment, the file is already closed!

More details/examples in OOPSLA 21, POPL 24.

Non-Escaping Scoped Capabilities

```
// library code
def withFile[T](path: String)(block: File => T◆): T◆ = { ... }

// user code
withFile("a.txt") { file =>
  ...
  { () => file.readLine() } // type error, cannot compile!
}
```

Reachability type system ensures that the returned value cannot capture "`file`":

$(\texttt{Unit => String})^{\texttt{file}} \not<: (\texttt{Unit => String})^{\blacklozenge}$

More details/examples in OOPSLA 21, POPL 24.

Destructive Effects: Affinity

- Affine effect: ensuring destructed resources are not used anymore
  - ownership transfer/move semantics

```
// ownership transfer:
var c1 = ...
var c2 = c1       // c1 and c2 aliased
val x = move(c1) // move semantics (e.g. in C++)
c1 += 1           // error
c2 += 1           // error, too!
```

More details/examples in OOPSLA 21, POPL 24.

Destructive Effects: Affinity

- Affine effect: ensuring destructed resources are not used anymore
  - ownership transfer/move semantics

effect store before move

$\{ \{c1, c2\} \rightarrow e \}$

```
// ownership transfer:
var c1 = ...
var c2 = c1         // c1 and c2 aliased
val x = move(c1)    // move semantics (e.g. in C++)
c1 += 1             // error
c2 += 1             // error, too!
```

More details/examples in OOPSLA 21, POPL 24.

Destructive Effects: Affinity

- Affine effect: ensuring destructed resources are not used anymore
  - ownership transfer/move semantics

effect gen by move

$\{ \{c1, c2\} \to e \} \blacktriangleright \{ \{c1\} \to \mathrm{Kill} \}$

```
// ownership transfer:
var c1 = ...
var c2 = c1       // c1 and c2 aliased
val x = move(c1) // move semantics (e.g. in C++)
c1 += 1           // error
c2 += 1           // error, too!
```

More details/examples in OOPSLA 21, POPL 24.

Destructive Effects: Affinity

- Affine effect: ensuring destructed resources are not used anymore
  - ownership transfer/move semantics

$$\{\ \{c1, c2\} \to e\ \} \blacktriangleright \{\ \{c1\} \to \text{Kill}\ \}$$
$$= \{\ \{c1, c2\} \cup \{c1\} \to e \blacktriangleright_{\text{PreEff}} \text{Kill}\ \}$$
$$= \{\ \{c1, c2\} \to \text{Kill}\ \}$$

```
// ownership transfer:
var c1 = ...
var c2 = c1         // c1 and c2 aliased
val x = move(c1)  // move semantics (e.g. in C++)
c1 += 1             // error
c2 += 1             // error, too!
```

More details/examples in OOPSLA 21, POPL 24.

Destructive Effects: Affinity

- Affine effect: ensuring destructed resources are not used anymore
  - ownership transfer/move semantics

$$\{\ \{c1, c2\} \rightarrow e\ \} \blacktriangleright \{\ \{c1\} \rightarrow \text{Kill}\ \}$$
$$= \{\ \{c1, c2\} \cup \{c1\} \rightarrow e \blacktriangleright_{\text{PreEff}} \text{Kill}\ \}$$
$$= \{\ \{c1, c2\} \rightarrow \text{Kill}\ \}$$

$$\{\ \{c1, c2\} \rightarrow \text{Kill}\ \} \blacktriangleright \{\ \{c1\} \rightarrow \text{Write}\ \}$$
$$= \{\ \{c1, c2\} \cup \{c1\} \rightarrow \text{Kill} \blacktriangleright_{\text{PreEff}} \text{Write}\ \}$$

undefined, compile-time error

```
// ownership transfer:
var c1 = ...
var c2 = c1        // c1 and c2 aliased
val x = move(c1)  // move semantics (e.g. in C++)
c1 += 1            // error
c2 += 1            // error, too!
```

More details/examples in OOPSLA 21, POPL 24.

Destructive Effects: Affinity

- Affine effect: ensuring destructed resources are not used anymore
  - ownership transfer/move semantics
  - one-shot continuation/effect handler
  - memory deallocation
  - message passing

```
// memory deallocation:
free(c1)
c1 += 1              // error

// message passing:
channel.send(c1)
c1 += 1              // error
```

```
// ownership transfer:
var c1 = ...
var c2 = c1       // c1 and c2 aliased
val x = move(c1)  // move semantics (e.g. in C++)
c1 += 1           // error
c2 += 1           // error, too!
```

More details/examples in OOPSLA 21, POPL 24.

Ongoing and Future Work

- Published work:
  - Polymorphic Reachability Types [POPL ‘24]
  - Graph IRs for Impure Higher-Order Languages – Making Aggressive Optimizations Affordable with Precise Effect Dependencies [OOPSLA ‘23]
  - Reachability Types [OOPSLA ‘21]

- Ongoing work:
  - Logic relations, soundness, termination, and equivalence
  - Algorithmic subtyping for escaping
  - Enabling reachability on cyclic data structures

- Possible future work:
  - Concurrency extension of the object language
  - Under-approximation reachability
  - Separation logic as the semantic interpretation

Summary

- Reachability Types: A family of type systems tracking aliases, separation, and effects for imperative higher-order languages

$$\Gamma^{\varphi} \vdash e : T^{q} \mid \varepsilon$$

Context: *what resources can be observed*
Space: *where are things/heap topology*
Time: *how things change by execution order*

- Rocq Mechanization & implementation:
https://github.com/TiarkRompf/reachability