Polymorphic Reachability Types

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

Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, Tiark Rompf
Purdue University, Galois Inc, Augusta University

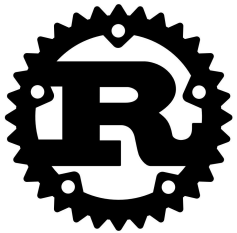
POPL 2024

Motivation

Memory safety, *thread safety, performance, …*

Secret sauce: ownership types

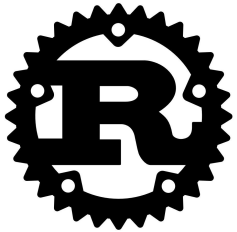
[Clarke et al., OOPSLA 98, Nobel et al. ECOOP 98]



Rust most admired language, Stack Overflow survey says

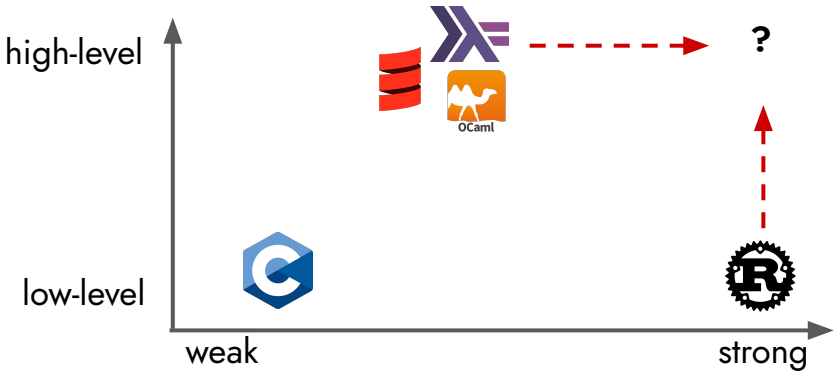
Stack Overflow 2023 Developer Survey finds that JavaScript and Python are the most used and most desired languages, but they fall far short of Rust in satisfying their users.

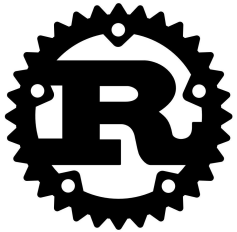
Motivation



What would post-Rust languages resemble?

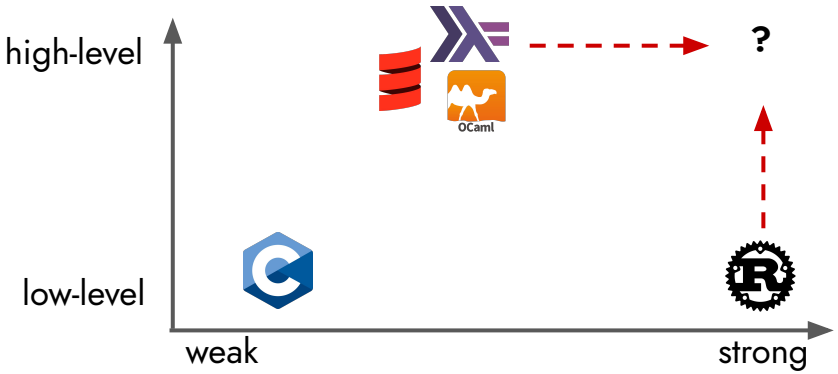
Motivation





What would post-Rust languages resemble?

Motivation



Ownership and Lifetimes

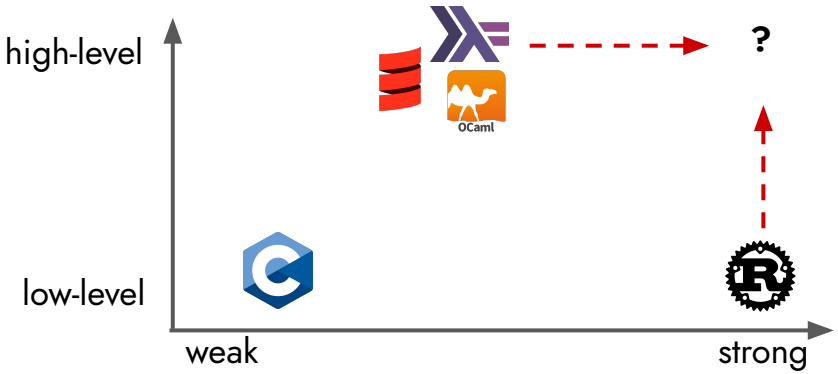
The ownership system is partially implemented, and is expected to get built out in the next couple of months. The basic support for ownership includes features like:

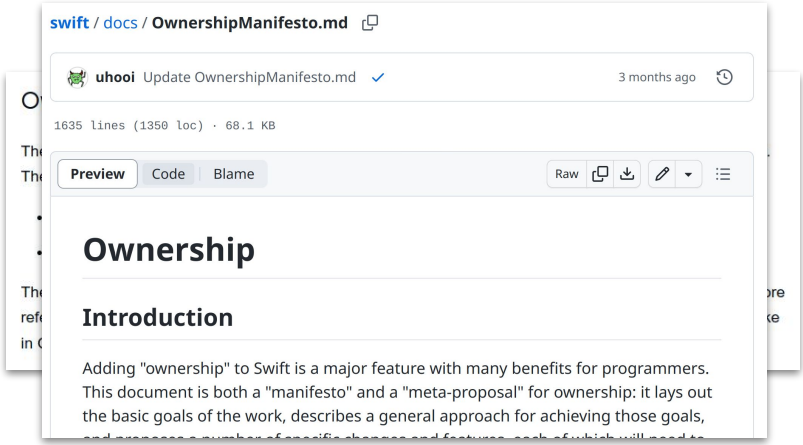
- Capture declarations in closures.
- Borrow checker: complain about invalid mutable references.

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

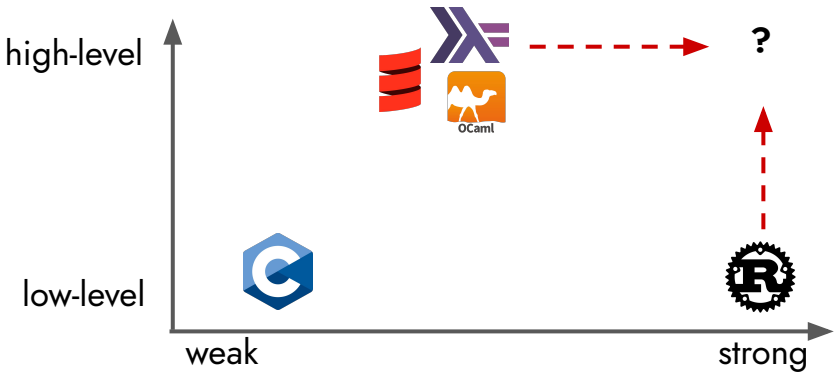
Motivation

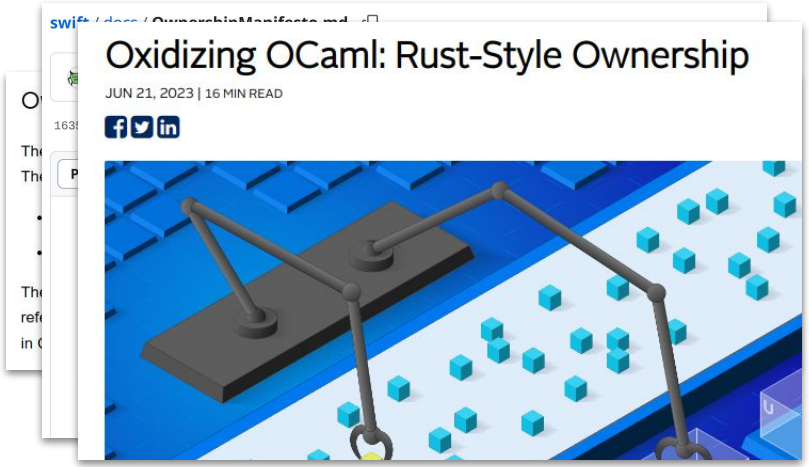




Swift / Apple

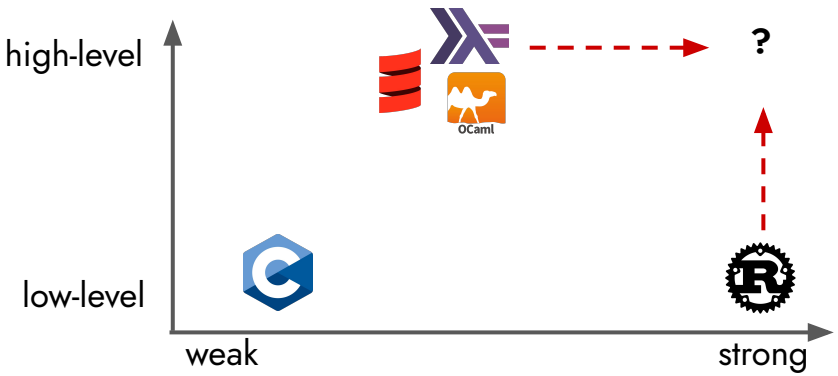
Motivation





OCaml / Jane Street

Motivation



Linear Haskell

Practical Linearity in a Higher-Order Polymorphic Language

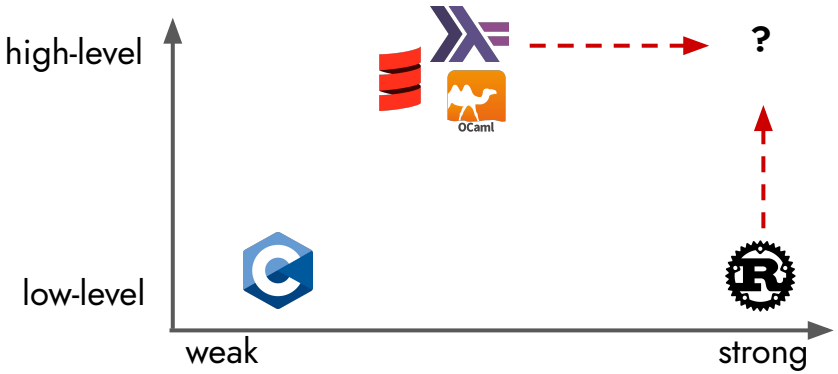
JEAN-PHILIPPE BERNARDY, University of Gothenburg, Sweden
MATHIEU BOESPFLUG, Tweag I/O, France
RYAN R. NEWTON, Indiana University, USA
SIMON PEYTON JONES, Microsoft Research, UK
ARNAUD SPIWACK, Tweag I/O, France

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

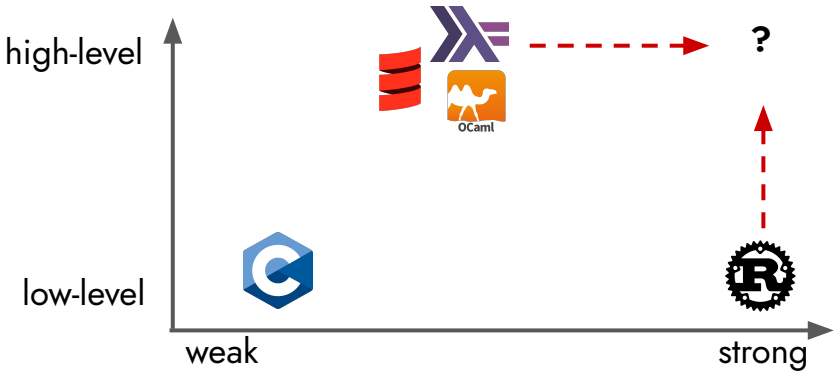
Type systems usually characterize the shape of values but not their free variables. However, many desirable safety properties could be guaranteed if one knew the free variables captured by values. We describe $CC_{<:\Box}$, a calculus where such captured variables are succinctly represented in types, 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 $CC_{<:\Box}$ 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



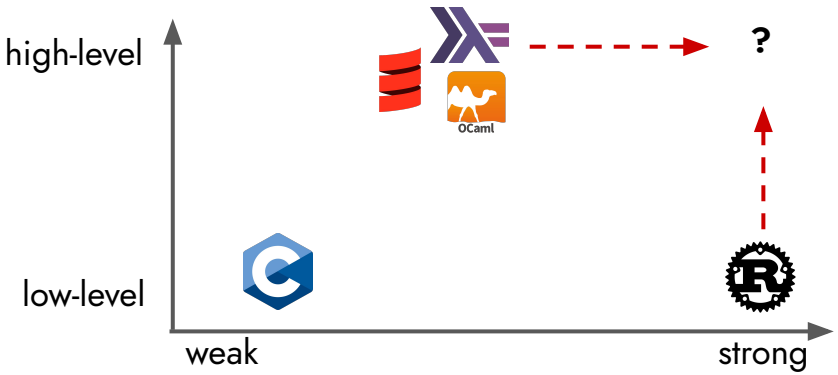
How to smoothly combine functional/type abstractions with resource tracking/control?

Rust's "*secret sauce*"

vs

Pervasive sharing from functional abstraction

Motivation



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

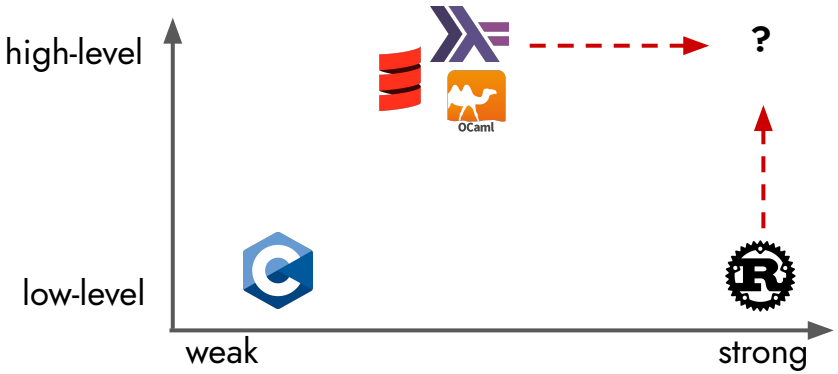
"shared XOR mutable"

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 …

Reachability Types

$$\Gamma \overset{\varphi}{\vdash} e : T \overset{q}{|} \varepsilon$$

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

- ***Context**: what capabilities can be observed*
  - lexical scope, capturing, escaping
- ***Space**: where are things/heap topology*
  - reachability, aliasing/sharing, separation
- ***Time**: how things change by execution order*
  - flow-sensitive effects, affinity, ownership transfer

Inspired by previous work of alias types, region-based type systems, effect systems, etc.

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

Reachability Types

$$\Gamma \overset{\varphi}{\vdash} e : T \overset{q}{|} \varepsilon$$

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

Example: Escaping pairs with shared mutable data

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

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

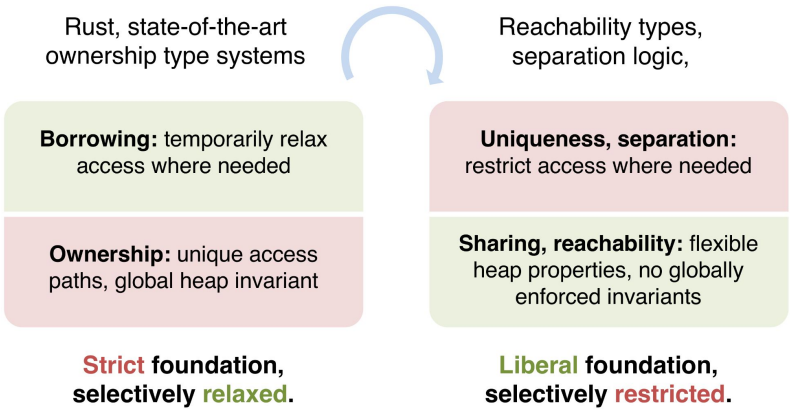
Reachability Types: tracking sharing and separation in higher-order languages
[OOPSLA 2021, Bao et al.]

Reachability Types ✓

 ✗

Reachability Types

$$\Gamma \overset{\varphi}{\vdash} e : T \overset{q}{|} \varepsilon$$

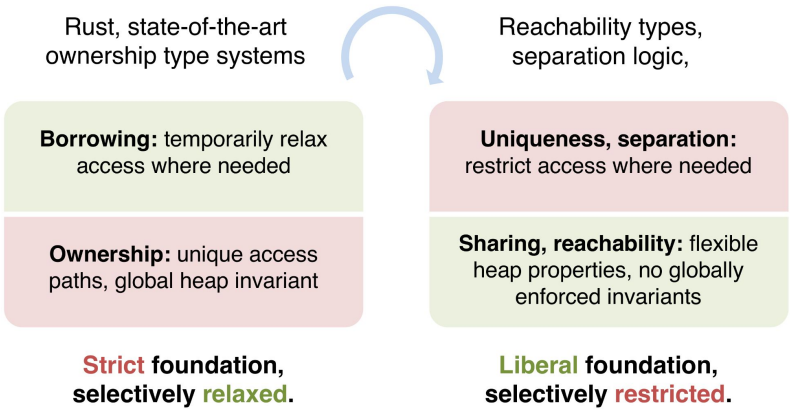


How to smoothly combine functional/type abstractions with resource tracking/control?

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

Polymorphic Reachability Types

$$\Gamma \overset{\varphi}{\vdash} e : T \overset{q}{|} \varepsilon$$



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

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

This work: smoothly combining reachability types with polymorphism

- ★ a new notion of freshness
- ★ precise lightweight reachability polymorphism
- ★ bounded type-and-reachability polymorphism

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

- Function types track the observable context:

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

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.

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

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

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.

- **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 without special treatment to distinguish it from untracked,
  or the system becomes non-parametric and leads to loss of precision (as in Bao et al.)

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.

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

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

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

  I.e. the universal/root capture set in Scala Capture Types, also need a special treatment to prevent “unboxing” the universal capability.

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

- More flexible and expressive compared with Scala Capturing Types:

```
try[Ref[Int]ᵀ](c => new Ref(42)) // not permitted to be “unboxed”
```

The Absence of Reachability

- In intersection type systems

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

- For reachability qualifiers, need to check stronger properties such as separation:

```
q1 ∩ q2 ⊆ ∅
```

The Absence of Reachability

- In intersection type systems

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

- For reachability qualifiers, need to check stronger properties such as separation:

```
q1 ∩ q2 ⊆ ⌀
```

- **Key Idea**: freshness marker as the argument qualifier

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

The Absence of Reachability

- In intersection type systems

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

- For reachability qualifiers, need to check stronger properties such as separation:

  $q1 \cap q2 \subseteq \varnothing$

- **Key Idea**: freshness marker as the argument qualifier

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

  Any argument is fresh for closed function `id`!

  Are there cases that we cannot apply some argument?

Checking Separation

- Applications check *observable separation* between the function and argument:

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

Checking Separation

- Applications check *observable separation* between the function and argument:

```
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 *observable separation* between the function and argument:

```
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 *observable separation* between the function and argument:

```
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} ⊆ ∅
```

Key idea: argument is fresh in *context* if the function can't *observe* overlap with other variables!

Checking Separation

- Applications check *observable separation* between the function and argument:

```
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} ⊆ ∅
```

  Key idea: argument is fresh in *context* if the function can't *observe* overlap with other variables!

- Function argument qualifier describes permissible overlap/aliasing patten:

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

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
}
```

Precise 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}
```

Result reachability can precisely depend on the argument reachability.

Precise 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}
```

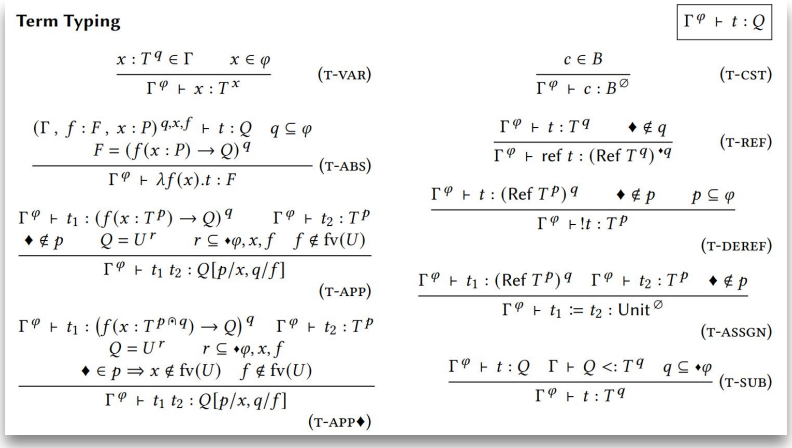
Result reachability can precisely depend on the argument reachability.

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

```
def id[Tz <: Top◆](x: T◆): T{x} = x
val p = makePair(a, b) // : Pair[Ref[Int]{a}, Ref[Int]{b}]
fst(p)                 // : Ref[Int]{a}
```

Formalization & Metatheory

- Simply-typed λ◆-calculus
- $F^{\blacklozenge}_{<:}$-calculus with bounded polymorphism
  - Typability of Church-encoding of pairs
- Syntactic soundness
  - Progress
  - Preservation: qualifiers may grow only due to freshness (new allocations)
- Preservation of separation: two separate terms remain separate after reduction steps.



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

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

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

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

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

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

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

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

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

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

Mechanization & Implementation

- Mechanized syntactic formalization in Coq
  - Alternative logical relation formalization in progress
- Prototype implementation Diamond language
  - Type checking of reachability types
- Both can be found at https://github.com/TiarkRompf/reachability

Contribution & Conclusion

- ★ Polymorphic reachability types
  - Tracking sharing/separation in higher-order generic languages
  - Representing freshness explicitly
  - Precise reachability polymorphism
  - F-sub style bounded polymorphism
- ★ Paves the way for integration of reachability types in practical impure functional languages.
- ★ Mechanization & prototype implementation: https://github.com/TiarkRompf/reachability

Contribution & Conclusion

- ★ Polymorphic reachability types
  - Tracking sharing/separation in higher-order generic languages
  - Representing freshness explicitly
  - Precise reachability polymorphism
  - F-sub style bounded polymorphism
- ★ Paves the way for integration of reachability types in practical impure functional languages.
- ★ Mechanization & prototype implementation: https://github.com/TiarkRompf/reachability

I'm on the job market!

More about my research:
https://continuation.passing.style/