Polymorphic Reachability Types

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

Guannan Wei[1] Oliver Bračevac[1,2] Songlin Jia[1] Yuyan Bao[3] Tiark Rompf[1]

[1] Purdue University / PurPL [2] Galois Inc. [3] Augusta University

Background: Reachability Types

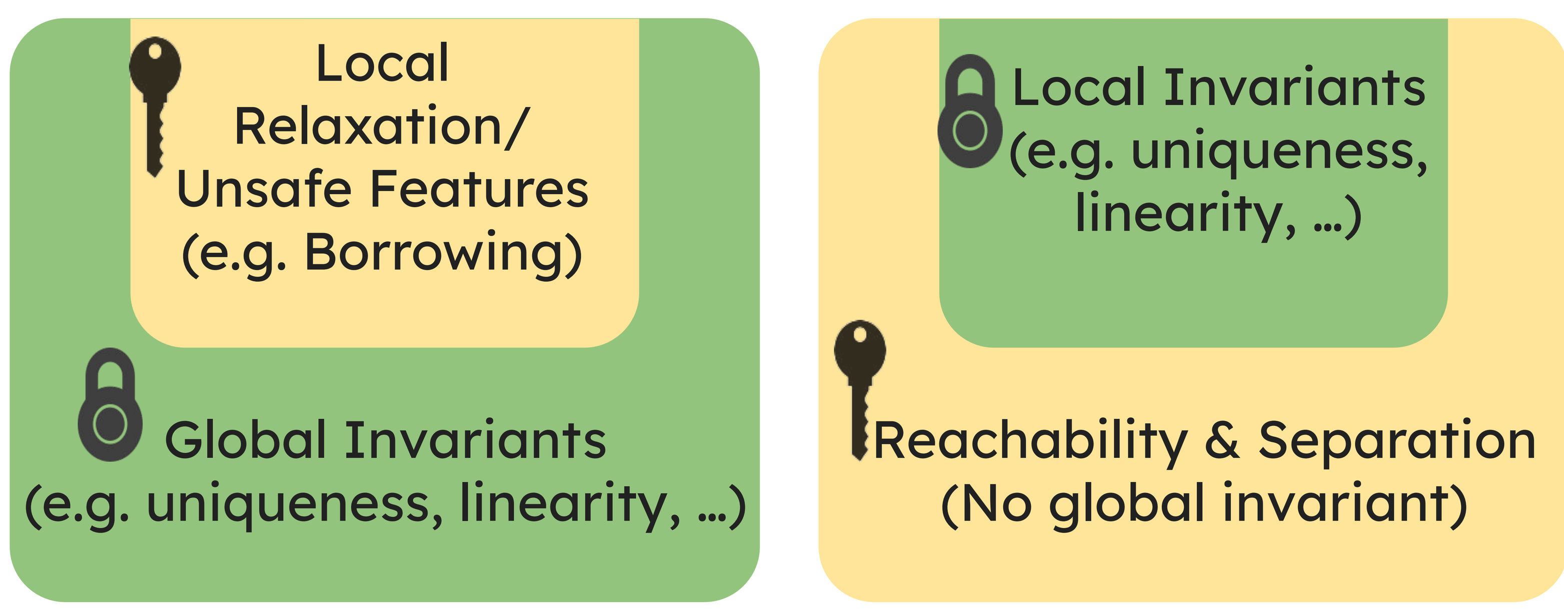
Reachability types (OOPSLA '21): tracking lifetimes, sharing, and separation in higher-order languages.

- Aiming at bringing Rust-style reasoning principles into higher-order functional languages.
- Challenge: side effects + pervasive sharing, capturing, and escaping in higher-order programs.
- Key idea: qualifying types with a set of variables

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

- Intuition: q is the set of variables that can be reached from the evaluation result of e .

Rust/ownership-style: Reachability types:



New: Tracking Freshness

- This work: use a special marker ◆ in qualifiers to explicitly track fresh resources that are not yet bound.

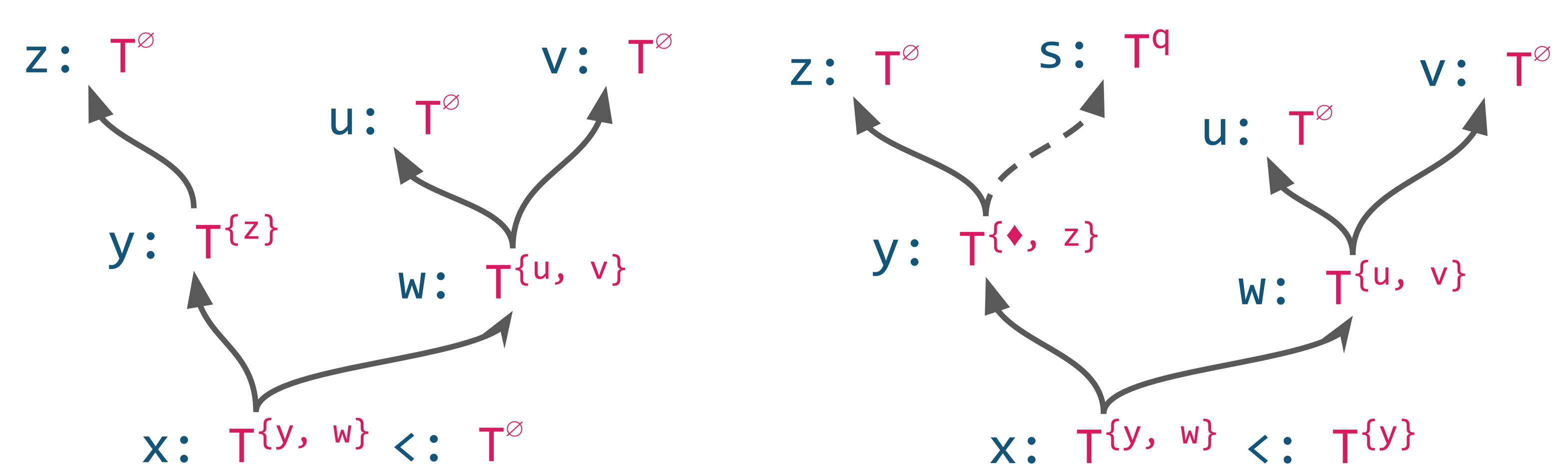
```
new Ref(42)        // : Ref[Int]^◆, fresh allocation
val i = 42         // : Int^∅, no tracking
```

Intuition: ◆ *represents a statically unknown set of reachable variables.*

- Preserving indirection in reachability tracking

```
val x = new Ref(42) // : Ref[Int]^x
val y = x           // : Ref[Int]^y
// in typing context y: Ref[Int]^x, x: Ref[Int]^◆
```

- Contextual subtyping of qualifiers



Upcasting allows *refining* qualifiers by looking up the context (left), but only up to the freshness marker (right).

New: Reachability Polymorphism

- This work: functions can abstract over the argument's qualifier and preserve *precise* reachability.
- Polymorphic identity function in F-sub style:

```
def id[T^z <: Top^◆](x: T^◆): T^x = x
def id[T](x: T^◆) = x // shorthand notation
```

- id is parametric over the argument reachability:

```
id(42)                // : Int^∅
val x = new Ref(42)
id(x)                 // : Ref[Int]^x
id(new Ref(42))       // : Ref[Int]^◆
```

- Qualifier-dependent applications:

```
... // c1: T^c1, c2: T^c2
def foo[T](x: T^{c1,◆}): T^x = { c1 := !c1 + 1; x }
// foo : ((x: T^{c1,◆}) => T^x)^c1
foo(c1) // : T^c1
foo(c2) // : T^c2 ← precision retained
```

Formalization & Metatheory

- Simply typed $\lambda^{\blacklozenge}$-calculus
- Parametric polymorphic $F_{<:}^{\blacklozenge}$-calculus
  - Bounded quantification over reachability qualifiers
- Type and qualifiers preservation: Qualifiers may increase only due to fresh allocations.
- Separation preservation: Two separate terms remain separate after reduction steps.

- Syntactic formalizations
  - Reachability Types (OOPSLA '21)
  - *Polymorphic Reachability Types (cond. acc. POPL '24)*
- Logical relation formalizations
  - Alias/effect-aware IR for optimizations (OOPSLA '23)
  - Allowing to prove termination, equivalence, etc.

Implementation

- Diamond prototype language
- Scala-like syntax in the frontend
- Type checker for polymorphic reachability types based on F-sub

Cool Examples

Sharing Mutable States (not expressible in Rust)

Self-references μp as upper bounds of reachability for the escaped pair:

```
def counter(n: Int) = {
  val c = new Ref(n)
  (() => c += 1, () => c -= 1)
}
// counter : Int =>
//    μp.Pair[(() => Unit)^p, (() => Unit)^p]^◆
val ctr = counter(0)
// ctr: Pair[(() => Unit)^ctr, (() => Unit)^ctr]^ctr
val incr = fst(ctr) // : (() => Unit)^ctr
val decr = snd(ctr) // : (() => Unit)^ctr
```

Safe Parallelization

Requiring two thunks that have disjoint qualifiers 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
}
```

More examples (borrowing, ownership transfer, capability programming, etc.) in OOPSLA 21 and our new preprint!