REACHABILITY TYPES

Tracking Aliasing and Separation in Higher-Order Functional Programs

Yuyan Bao[1] Guannan Wei[2]◆ Oliver Bračevac[2]◆ Luke Jiang[2] Qiyang He[2] Tiark Rompf[2]

[1] University of Waterloo [2] Purdue University / PurPL ◆ In-person poster presenter ◆ Virtual poster presenter

Motivation

Reachability types: Expressive ownership-style reasoning for higher-order functional languages.

Popular ownership models:

- Local Relaxation/ Unsafe Features (e.g. Borrowing)
- Global Invariants (e.g. uniqueness, linearity, …)

This work flips it on its head:

- Local Invariants (e.g. uniqueness, linearity, …)
- Separation & Reachability (No global invariant)

State-of-the-art ownership systems restrict the use of higher-order functions. Consider the "counter" program that can be elegantly implemented in functional languages:

```
def counter(n: Int) = {
  val c = new Ref(n)
  (() => c += 1, () => c -= 1)
}
val (incr, decr) = counter(0)
incr(); incr(); decr()
```

However, one has to use dynamic reference counting in Rust, by the "shared XOR mutable" restriction. How can we remove such restrictions and enable safe & expressive ownership-style type systems for higher-order languages?

```
fn counter(n: i64) -> (impl Fn()->(),
                       impl Fn()->()) {
  let c = Rc::new(Cell::new(n));
  let c1 = c.clone();
  let c2 = c.clone();
  (move || { c1.set(c1.get() + 1); },
   move || { c1.set(c2.get() - 1); })
}
```

Separation logics have been established as the formal foundation for Rust-style ownership type systems (e.g. RustBelt), what if we build a separation substrate into user-level syntactic types?

Reachability Types, Informally

- Tracking reachable variables at the type level:

```
new Ref(42)          // : Ref[Int]^∅
val x = new Ref(42)  // : Ref[Int]^{x}
val y = x            // : Ref[Int]^{x, y}
val i = 42           // : Int^⊥, no tracking
```

- Function types track the free variables:

```
val c1: Ref[Int]^{c1}; val c2: Ref[Int]^{c2}
def addRef(c: Ref[Int]^∅) = c1 := !c1+!c; c1
// addRef: (Ref[Int]^∅ => Ref[Int]^{c1})^{c1}
```

observable aliases of argument by the function body

free variables of the function

- Applications check if argument is aliased with function's qualifier, guaranteeing *observable separation*:

```
addRef(c1) // type error because {c1}⊓{c1}!=∅
addRef(c2) // ok because {c2}⊓{c1}=∅
```

- Function domain controls permissible overlap:

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

- How should we type escaping functions?

```
{ () => new Ref(0) }
// (() => Ref[Int]^∅)^∅ ~> (() => Ref[Int]^∅)^∅
{ val y = new Ref(0); () => !y }
// (() => Int^⊥)^{y} ~> (() => Int^⊥)^∅
{ val y = new Ref(0); () => y }
// (() => Ref[Int]^{y})^{y} ~> removing y is wrong
```

Use DOT-style self-reference for functions:

```
   f(() => Ref[Int]^{y})^{y}
<: f(() => Ref[Int]^{f})^{y}  // self abstraction
~> f(() => Ref[Int]^{f})^∅    // now well-formed
```

Formalization: λ^*

Typing judgment $\Gamma \vdash t : T^q$

Qualifiers $q \in \{\bot\} \cup \mathcal{P}_{\text{fin}}(\text{Var})$

- Reachability type system λ^* based on STLC.
- *Type and qualifiers preservation*: Qualifiers may increase only due to fresh allocations.
- *Separation preservation*: Two separate terms remain separate after reduction steps.

Reachability-and-Effect System

- Tracking precise aliasing-aware effects by combining reachability types with effect quantales [Gordon 2021]: $\Gamma \vdash t : T^q \mid \epsilon$
- A flow-insensitive instantiation that enables finer-grained non-interference with read/write effects.
- A flow-sensitive instantiation that models ownership transfer, move semantics, unique references, nested references, etc.
- All we need is to use flow-sensitive "**kill**" effects that disable further accesses of a value and its aliases:

```
def f(x: Ref[Int]^∅) = { val y = move(x); ... }
val z = new Ref(0)
f(z) // now z is killed by move
!z   // type error
```

- Case studies (more details in the paper!)
  - Algebraic Effect and Handlers
  - Control Operators
  - Safe Concurrency Combinators