REACHABILITY TYPES

Tracking Aliasing and Separation in Higher-Order Functional Programs

SPLASH/OOPSLA 2021

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

OWNERSHIP TYPE SYSTEMS

The “Shared XOR Mutable” Principle

OWNERSHIP TYPE SYSTEMS

Higher-Order Functions: "Counter" Examples

A Counter in { Scheme, ML, Scala,...} :

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

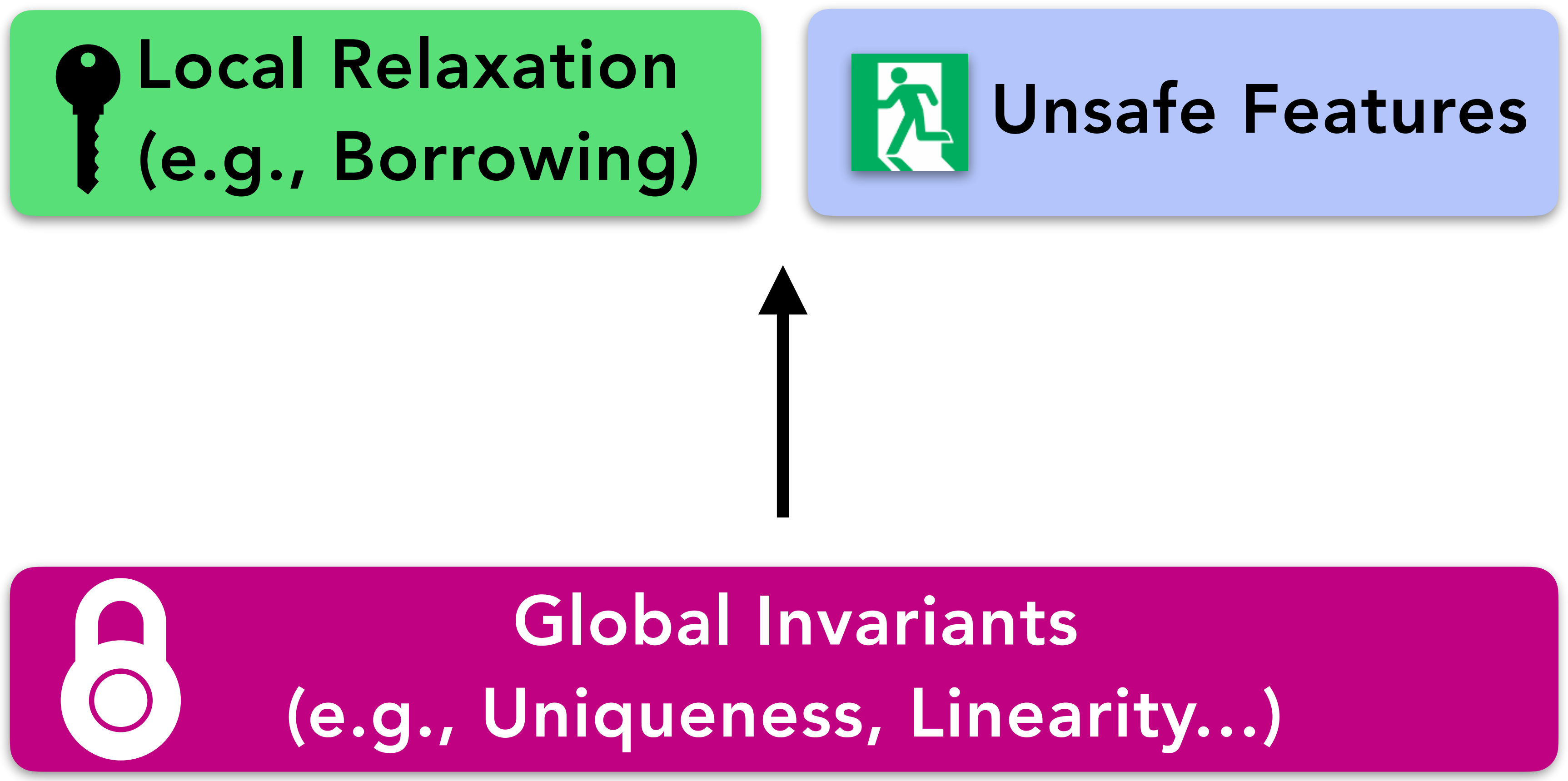
Let's Make One in Rust :

```
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); })
}
```

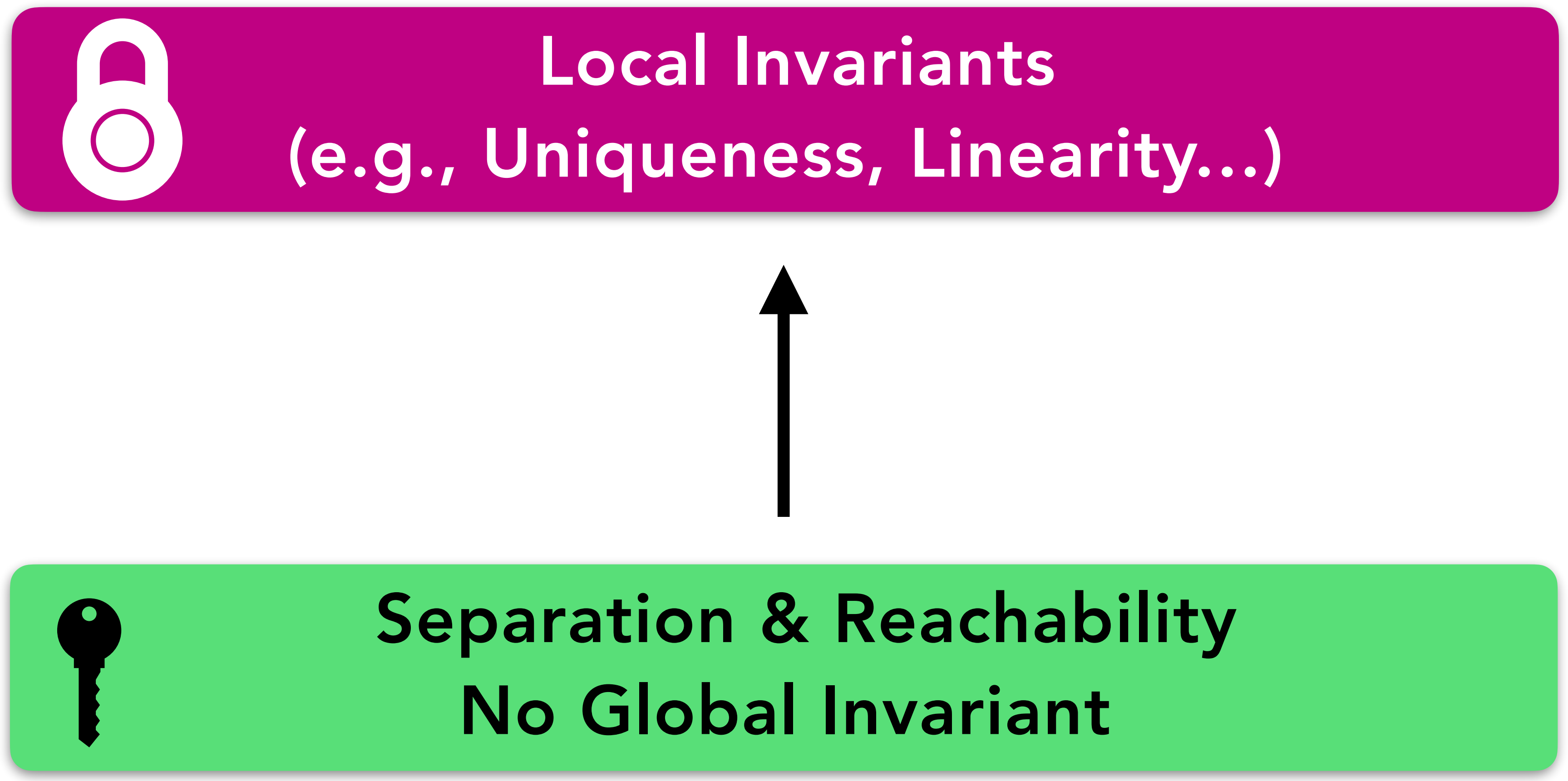
Dynamic reference counting, no static lifetime tracking!

OWNERSHIP TYPE SYSTEMS

The Prevailing Ownership and Borrowing Model:



This Work Flips it on its Head:



RETHINKING OWNERSHIP IN TERMS OF SEPARATION LOGIC

RustBelt: Securing the Foundations of the Rust Programming Language

RALF JUNG, MPI-SWS, Germany
JACQUES-HENRI JOURDAN, MPI-SWS, Germany
ROBBERT KREBBERS, Delft University of Technology, The Netherlands
DEREK DREYER, MPI-SWS, Germany

- What happens if we expose separation and overlap all the way to the user-facing types?
- We get expressive ownership-style reasoning across higher-order functions!

I

REACHABILITY TYPES

THE λ^* CALCULUS

```
new Ref(42)               : Ref[Int]^∅

val x = new Ref(42)       : Ref[Int]^{x}

val i = 42                : Int^⊥

val y = x                 : Ref[Int]^{x,y}

val z = !y                : Int^⊥

x := 0                    : Unit^⊥
```

Intuition: Reachability Types & Qualifiers

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

$$q \in \{ \perp \} \uplus \mathcal{P}_{\mathrm{fin}}(\mathrm{Var})$$

Computation t yields a *T* value which may reach all variables in *q*.

$\perp$ is untracked (often omitted).

$\varnothing$ means "fresh", no sharing w. context.

A simply-typed lambda calculus (STLC) with qualifiers, mutable references, recursion, and subtyping.

FUNCTIONS

Qualifiers Track Free Variables

```
val c1 : Ref[Int]{c1}; val c2 : Ref[Int]{c2}

def addRef(c3 : Ref[Int]∅) =     // (Ref[Int]∅ => Ref[Int]{c1}){c1}
  c1 := !c1 + !c3; c1

addRef(c2)  // ok
addRef(c1)  // type error
```

addRef's implementation
reaches/closes over c1.

addRef's implementation
must not share aliasing
with its argument: $\varnothing \sqcap \{c_1\} = \varnothing$

```
def addRef2(c3 : Ref[Int]{c1}) =
  c1 := !c1 + !c3; c1

addRef2(c1) // ok now

// (Ref[Int]{c1} => Ref[Int]{c1}){c1}
```

Intuition: Observable Separation

- Functions track their free variables, consistent with view as closure records.
- To prevent interference from uncontrolled aliasing, functions are separated from their arguments
- If full separation is too strict, we may adjust the function domain's qualifier for degrees of overlap.

ESCAPING CLOSURES

How Can We Track their Free Variables?

Type Assignment Inside vs. Outside of Lexical Scopes

`{ () => new Ref(42) }` : (() => Ref[Int]$^{\varnothing}$)$^{\varnothing}$ ~> (() => Ref[Int]$^{\varnothing}$)$^{\varnothing}$

`{ val y = new Ref(42); () => !y }` : (() => Int)$^{\{y\}}$ ~> (() => Int)$^{\varnothing}$

`{ val y = new Ref(42); () => y }` : (() => Ref[Int]$^{\{y\}}$)$^{\{y\}}$ ~> what now?

Wrong: (() => Ref[Int]$^{\varnothing}$)$^{\varnothing}$ returns a ***fresh*** reference on each call!

Right:

f(() => Ref[Int]$^{\{y\}}$)$^{\{y\}}$

<: f(() => Ref[Int]$^{\{f\}}$)$^{\{y\}}$

~> f(() => Ref[Int]$^{\{f\}}$)$^{\varnothing}$

Intuition: Function Self-Qualifiers

- Abstract over the free variables by letting a function type refer to itself. A concept borrowed from **DOT/Scala**!
- The self-qualifier's presence indicates that *some* qualifier escapes (existential statement).
- Subtyping (<:) makes their use ergonomic, compared to existential types.

LIGHTWEIGHT REACHABILITY POLYMORPHISM

Dependent function type!

```
def inc(x : Ref[Int]^∅) = { x := !x + 1; x } // : ((x : Ref[Int]^∅) => Ref[Int]^{x})^⊥
```

Lightweight Polymorphism (No Quantifiers!)

```
val c : Ref[Int]^{a,b,c} ; val d : Ref[Int]^{d}

inc(c)            // : Ref[Int]^{a,b,c}

inc(d)            // : Ref[Int]^{d}

inc(new Ref(0))   // : Ref[Int]^∅
```

Full details in the paper!

$$\begin{array}{lcl} t & ::= & c \mid x \mid \lambda f(x).t \mid t_1\ t_2 \mid \mathsf{ref}\ t \mid !t \mid t_1 := t_2 \\ T & ::= & B \mid \mathsf{Ref}\ T \mid f(x : T^q) \to T^q \\ \Gamma & ::= & \varnothing \mid \Gamma, x : T^q \end{array} \qquad \begin{array}{lcl} x, y, \ldots, f, g, \ldots & \in & \mathsf{Var} \\ \alpha, \beta, \gamma & \in & \mathcal{P}_{\mathrm{fin}}(\mathsf{Var}) \\ q & ::= & \bot \mid \alpha \end{array}$$

Fig. 2. The syntax of λ^*.

$\boxed{\Gamma \vdash t : T^q}$

$$\textsc{T-cst}\ \frac{c \in B}{\Gamma \vdash c : B^{\bot}} \qquad \textsc{T-var}\ \frac{\Gamma(x) = T^q}{\Gamma \vdash x : T^q} \qquad \textsc{T-ref}\ \frac{\Gamma \vdash t : T^{\bot}}{\Gamma \vdash \mathsf{ref}\ t : (\mathsf{Ref}\ T)^{\varnothing}} \qquad \textsc{T-assign}\ \frac{\Gamma \vdash t_1 : (\mathsf{Ref}\ T)^q \quad \Gamma \vdash t_2 : T^{\bot}}{\Gamma \vdash t_1 := t_2 : \mathsf{Unit}^{\bot}} \qquad \textsc{T-deref}\ \frac{\Gamma \vdash t : (\mathsf{Ref}\ T)^q}{\Gamma \vdash !t : T^{\bot}}$$

$$\textsc{T-abs}\ \frac{F = f(x : T_1^{q_1}) \to T_2^{q_2} \quad (\Gamma, f : F^{q_f + f}, x : T_1^{q_1 + x})^{q_f \sqcup \{f,x\}} \vdash t : T_2^{q_2}}{\Gamma \vdash \lambda f(x).t : F^{q_f}} \qquad \textsc{T-app}\ \frac{\Gamma \vdash t_1 : (f(x : T_1^{q_1 \sqcap q_f}) \to T_2^{q_2})^{q_f} \quad \Gamma \vdash t_2 : T_1^{q_1} \quad x, f \notin FV(T_2)}{\Gamma \vdash t_1\ t_2 : T_2^{q_2[q_1/x,\ q_f/f]}} \qquad \textsc{T-sub}\ \frac{\Gamma \vdash t : T_1^{q_1} \quad \Gamma \vdash T_1^{q_1} <: T_2^{q_2}}{\Gamma \vdash t : T_2^{q_2}}$$

$\boxed{\Gamma \vdash T_1^{q_1} <: T_2^{q_2}}$

$$\frac{\Gamma \vdash q_1 <: q_2 \quad \Gamma \vdash T_1^{\bot} <: T_2^{\bot} \quad \Gamma \vdash T_2^{\bot} <: T_1^{\bot}}{\Gamma \vdash (\mathsf{Ref}\ T_1)^{q_1} <: (\mathsf{Ref}\ T_2)^{q_2}}\ \textsc{S-ref} \qquad \frac{\Gamma \vdash q_1 <: q_2}{\Gamma \vdash B^{q_1} <: B^{q_2}}\ \textsc{S-base}$$

$$\frac{\Gamma \vdash q_5 <: q_6 \quad \Gamma \vdash T_3^{q_3} <: T_1^{q_1} \quad \Gamma, f : (f(x : T_1^{q_1}) \to T_2^{q_2})^{q_5 + f}, x : T_3^{q_3 + x} \vdash T_2^{q_2} <: T_4^{q_4}}{\Gamma \vdash (f(x : T_1^{q_1}) \to T_2^{q_2})^{q_5} <: (f(x : T_3^{q_3}) \to T_4^{q_4})^{q_6}}\ \textsc{S-fun}$$

Fig. 3. Typing and subtyping rules of λ^*.

$$\begin{array}{lcl} C & ::= & \square \mid C\ t \mid v\ C \mid \mathsf{ref}\ C \mid !C \mid C := t \mid v := C \\ v & ::= & \lambda f(x).t \mid c \mid l \mid \mathsf{unit} \\ t & ::= & \ldots \mid l \end{array} \qquad \begin{array}{lcl} l & \in & \mathsf{Loc} \\ \sigma & ::= & \varnothing \mid \sigma, l \mapsto v \end{array}$$

$\boxed{t \mid \sigma \to t' \mid \sigma'}$

$$\begin{array}{lcll} C[(\lambda f(x).t)\ v] \mid \sigma & \to & C[t[v/x, (\lambda f(x).t)/f]] \mid \sigma & & [\beta] \\ C[\mathsf{ref}\ v] \mid \sigma & \to & C[l] \mid (\sigma, l \mapsto v) & l \notin dom(\sigma) & [\textsc{Ref}] \\ C[!l] \mid \sigma & \to & C[\sigma(l)] \mid \sigma & l \in dom(\sigma) & [\textsc{Deref}] \\ C[l := v] \mid \sigma & \to & C[\mathsf{unit}] \mid \sigma[l \mapsto v] & l \in dom(\sigma) & [\textsc{Assign}] \end{array}$$

Fig. 4. Reduction Semantics of λ^*.

TYPE SOUNDNESS

Progress & Preservation [Wright & Felleisen '94]

Preservation

If $\varnothing \mid \Sigma \vdash \sigma$, $\varnothing \mid \Sigma \vdash t : T^{q}$, and $t \mid \sigma \longrightarrow t' \mid \sigma'$,

then $\varnothing \mid \Sigma' \vdash \sigma'$ and $\varnothing \mid \Sigma' \vdash t' : T^{q \oplus q'}$

for some $\Sigma' \supseteq \Sigma$ and $q' \sqsubseteq dom(\Sigma') \backslash dom(\Sigma)$

Corollary: Preservation of Separation

$$\begin{array}{ccccc} \varnothing \mid \Sigma \vdash t_1 : S^{q_1} & \parallel & \varnothing \mid \Sigma \vdash t_2 : T^{q_2} & & q_1 \sqcap q_2 \sqsubseteq \varnothing \\ \downarrow & & \downarrow & & \\ \varnothing \mid \Sigma' \vdash t_1' : S^{q_1'} & \parallel & \varnothing \mid \Sigma' \vdash t_2' : T^{q_2'} & & q_1' \sqcap q_2' \sqsubseteq \varnothing \end{array}$$

- Information may increase due to fresh allocations.
- Cancelling union ensures that untracked terms remain untracked: $\bot \oplus q = \bot \quad \alpha \oplus q = \alpha \sqcup q$
- Limitation: References must be *shallow*. We will solve this next.

- Interleaving two computations with separate answers keeps them separate.
- Reduction steps never introduce spurious aliasing/sharing between the two answers.

HIGHER-ORDER FUNCTIONS

Non-Escaping Values [Osvald et al. 2016]

```
def try[A⌀](block: (CanThrow⌀ => A⌀)⌀): Option[A]⌀
```

Return value cannot
close over the capability.

```
val c1 = new Ref(0)

try { throw =>

  c1 += 1

  if (error) throw(new Exception("legal"))

  () => throw(new Exception("illegal")

}
```

- The base calculus supports effects as capabilities models and lightweight effect polymorphism [Brachthäuser et al. 2020].
- Reachability types alone do not capture linear consumption of capabilities, etc. This requires a proper effect system.

Non-Interference

```
def par(a: (() => Unit)⌀)(b: (() => Unit)⌀): Unit
```

Threads must have
non-overlapping qualifiers

```
val c1 = new Ref(0); val c2 = new Ref(0)
// ok, no overlap
par { c1 := !c1 + 1 } { c2 := !c2 + 2 }
// type error, overlapping
par { c1 := !c1 + !c2 } { c2 := !c1 + !c2 }
// type error, overlapping, but safe (!)
par { !c1 + !c2 } { !c1 + !c2 }
```

- Effect systems can help making more fine-grained distinctions.

II

REACHABILITY & EFFECTS

REACHABILITY-AND-EFFECT SYSTEMS

- Reachability sets permit very precise effect systems, at the granularity of variables, in both flow-insensitive and flow-sensitive flavors.
- Effects make reachability types powerful enough to enable ownership transfer, consumption policies (e.g., linearity), borrowing, etc.
- All we need are flow-sensitive “kill” effects to model nested references, consumption policies, move semantics, etc.

FLOW-INSENSITIVE EFFECTS

Example: Finer-grained Non-Interference with Read/Write Effects

```
par { !c1 + !c2 }    { c4 := !c1 + !c2 + !c3 }

    ({c1,c2} : rd)       ({c1,c2,c3} : rd, {c4} : wr)

                rd ⊔ rd = rd ✓
```

FLOW-SENSITIVE KILL EFFECTS

Enable Uniqueness, Linearity, Ownership Transfer & More

Example: Use-Once Functions from Self-Killing

```
// fun(Int =>({fun} : kill) String)∅
def fun(x) = { “Goodbye, cruel world!” }

fun(0) // fun at most once
fun(1) // type error, no more fun!
```

RECOVERING NESTED REFERENCES

Move Semantics and Ownership Transfer via Kill Effects

```
def f(x: Ref[Int]ø) = { val y = move(x); … }

val z = new Ref(1)
f(z) // z is killed by f and unusable
!z   // type error
```

CASE STUDIES IN THE PAPER

Reachability Types and Flow-(in)sensitive Effects for:

- Control Operators
- Algebraic Effects and Handlers
- Concurrency Combinators

$$\frac{\Gamma, k : \left(k(x : A^{q_1}) \to^{(\epsilon_k \rhd KE)} B\right)^{\{k\}} \vdash t : A^{q_2} \mid \epsilon \quad NE}{\Gamma \vdash \mathscr{C}\ k \text{ in } t : A^{q_1} \mid \epsilon}$$

Variants

let/cc: $B = \text{Nothing}^{\perp}$ shift: $B = C^{q_3}$

Attributes

Escaping (yes/no): $NE = \text{true}$ $NE = k \notin FV(A^{q_2})$

Affine (yes/no): $KE = \{(\{k\}, \text{kill})\}$ $KE = \{\}$

EFFECT QUANTALES

Polymorphic Iterable Sequential Effect Systems

COLIN S. GORDON, Drexel University

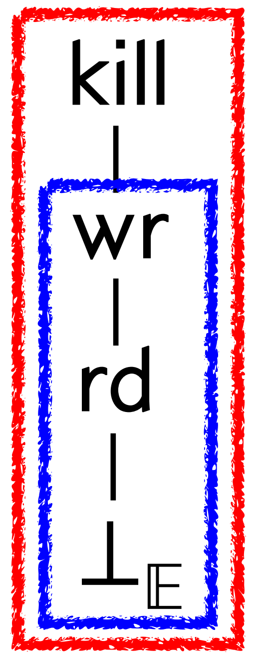
Effect Quantale [Gordon 2021]:

A structure $(\mathbb{E}, \sqcup, \rhd, I)$ where
$(\mathbb{E}, \sqcup)$ is a *partial join semi lattice*, and
$(\mathbb{E}, \rhd, I)$ is a *partial monoid.*

Store-Sensitive Effect Quantale (New Here):

The lifting of a quantale $(\mathbb{E}, \sqcup, \rhd, I)$
to a quantale over disjoint finite maps $\{\overline{(\alpha, \epsilon_{\mathbb{E}})}\}$,
assigning effects to reachability sets.

Example Effect Quantale:



Flow *sensitive*

Flow *insensitive*

$\rhd$	$\perp_{\mathbb{E}}$	rd	wr	kill
$\perp_{\mathbb{E}}$	$\perp_{\mathbb{E}}$	rd	wr	kill
rd	rd	rd	wr	kill
wr	wr	wr	wr	kill
kill	undefined			

SUMMARY & CONTRIBUTIONS

Reachability Types

- Ownership-style reasoning for impure higher-order functional programs.
- Track sharing and its absence, inspired by separation logic.
- No global heap invariants.
- Statically safe, lightweight types & idiomatic code.

Reachability-and-Effect System

- Extensible effect system, based on store-sensitive effect quantales.
- All we need are flow-sensitive “kill” effects to model nested references, linearity, uniqueness, move sematics, etc.

Artifacts

- Interactive Prototype.
- Coq mechanization of variants of the base λ^* calculus.
- Available at:

  `https://github.com/TiarkRompf/reachability`