DIAMONDS AND RUST

Oliver Bračevac

PurPL Seminar
10-02-2023

I

REACHABILITY TYPES

SEAMLESS OWNERSHIP FOR IMPURE FUNCTIONAL LANGUAGES

OWNERSHIP TYPE SYSTEMS

The “Shared XOR Mutable” Principle in Rust

OWNERSHIP TYPE SYSTEMS

Do Not Scale to Higher-Level Functional Languages!

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

How Can We Make Them Scale?

Reachability Types: Tracking Aliasing and Separation in Higher-Order Functional Programs

YUYAN BAO, University of Waterloo, Canada
GUANNAN WEI, Purdue University, USA
OLIVER BRAČEVAC, Purdue University, USA
YUXUAN JIANG, Purdue University, USA
QIYANG HE, Purdue University, USA
TIARK ROMPF, Purdue University, USA

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.

Lets Flip it on its Head with Reachability Types & Separation Logic!

Uniqueness, separation: restrict access where needed

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

Liberal foundation, selectively restricted.

REACHABILITY IN 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

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

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

Computation t yields a *T* value which <u>may</u> 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

addRef's implementation ***reaches/closes over*** `c1`.

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

```
addRef(c2)  // ok

addRef(c1)  // type error


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

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

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

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

Right:

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

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

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

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.

HIGHER-ORDER FUNCTIONS

Non-Escaping Values [Osvald et al. 2016]

```
def try[A^∅](block: (CanThrow^∅ => A^∅)^∅): 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.

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!)

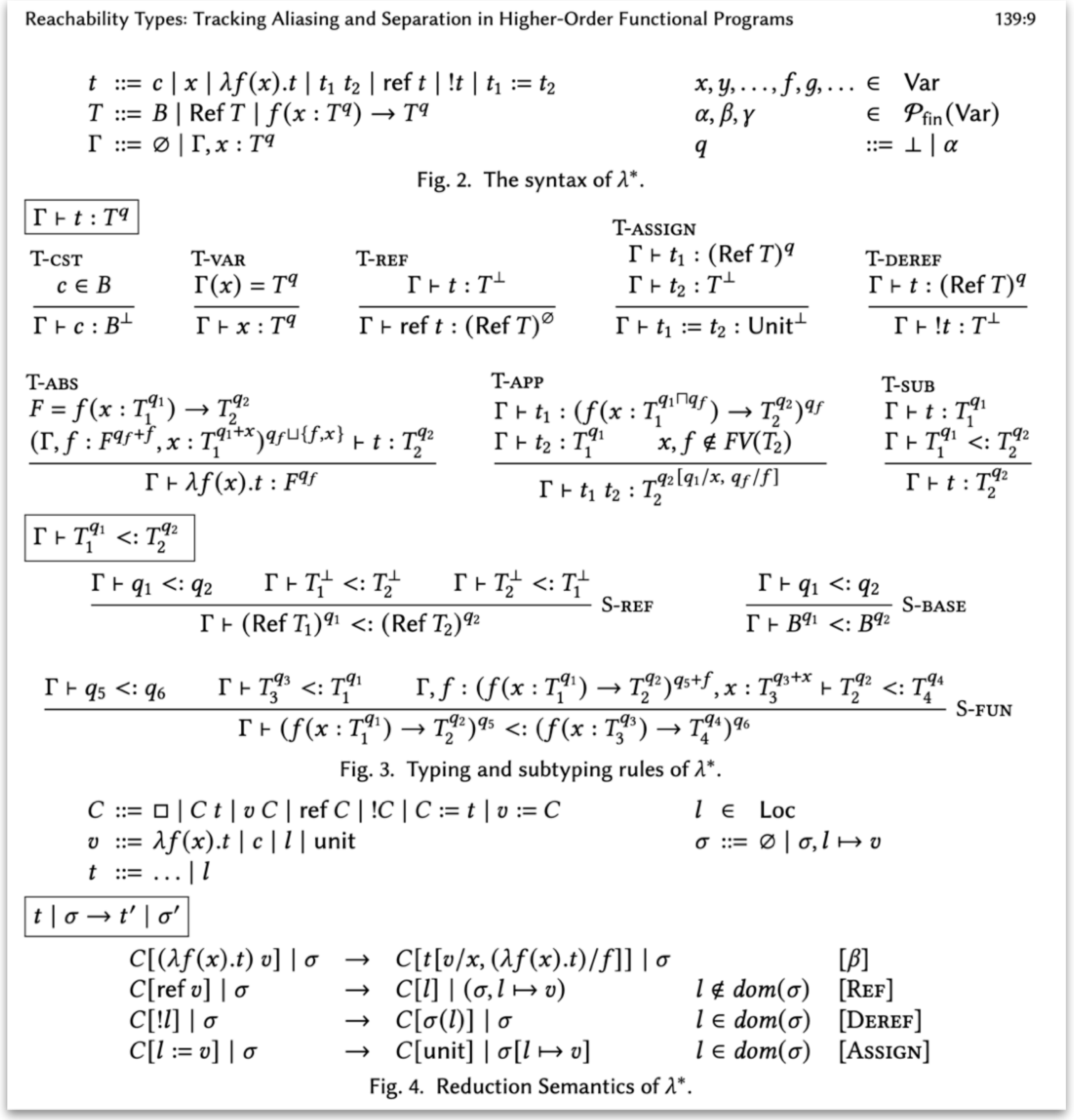
```
val c : Ref[Int]^{a,b,c} ; val d : Ref[Int]^{d}

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

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

inc(new Ref(0))     // : Ref[Int]^∅
```

Full details in the OOPSLA'21 paper!



Reachability Types: Tracking Aliasing and Separation in Higher-Order Functional Programs 139:9

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

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^*.

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

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

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

Fig. 4. Reduction Semantics of λ^*.

TYPE SOUNDNESS

Progress & Preservation [Wright & Felleisen '94], Mechanized in Coq

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 $q' \sqsubseteq dom(\Sigma') \backslash dom(\Sigma)$

- 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.

Corollary: Preservation of Separation

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

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

REACHABILITY-AND-EFFECT SYSTEMS

- We get **a lot of mileage** just from reachability + overlap checking, **at the price of** prohibiting nested references of the form `Ref[Ref[…]]`.
- Reachability sets permit very precise effect systems, at the granularity of variables, in both flow-insensitive and flow-sensitive flavors.
- All we need are flow-sensitive “kill” effects to recover 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  ✔
```

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

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

             rd ⊔ wr = wr  ☠️
```

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

```
// f((x:Ref[Int]^∅) =>^({x} : kill) T)^q
def f(x: Ref[Int]^∅) = { val y = move(x); … }

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

EFFECT QUANTALES

Polymorphic Iterable Sequential Effect Systems

COLIN S. GORDON, Drexel University

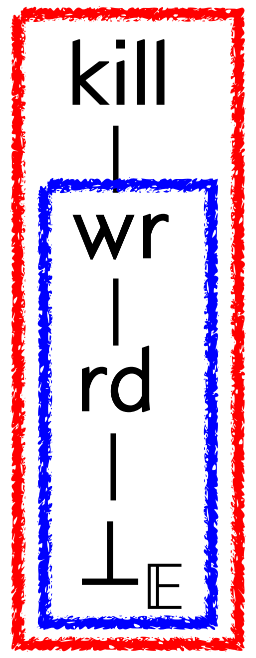
Effect Quantale [Gordon 2021]:

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

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

INTERIM CONCLUSION

Seamless & scalable Rust-style systems can be achieved in impure higher-order languages!

All you need is a little shift in perspective:

Uniqueness, separation:
restrict access where needed

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

Liberal foundation,
selectively restricted.

II

POLYMORPHISM AND DATA TYPES

```
def try[A^∅](block: (CanThrow^∅ => A^∅)^∅): A^∅
```

🤔 **Can we be polymorphic in qualifiers and types at the same time?**

REACHABILITY POLYMORPHISM REVISITED

```
def id(x: T^∅): T^{x} = x

val x: T^{x,a,b} = …;    val y: T^{y,z} = …

id(x) // : T^{x}[x ↦ {x,a,b}] = T^{x,a,b}

id(y) // : T^{x}[x ↦ {y,z}]   = T^{y,z}
```

REACHABILITY POLYMORPHISM REVISITED

def `id(x:` $T^{\varnothing}$`):` $T^{\{x\}}$ `= x`

val `i:` $T^{\bot}$ `= …`

`id(i) //` $: T^{\{x\}[x \mapsto \bot]} = T^{\varnothing}$ 😲

def `id'(x:` $T^{\bot}$`):` $T^{\bot}$ `= x`

`id'(i) //` $: T^{\bot[x \mapsto \bot]} = T^{\bot}$

Suppose $\{x\}[x \mapsto \bot] = \bot$

def `fakeid(x:` $T^{\varnothing}$`):` $T^{\{x\}}$ `= alloc()`

`fakeid(i) //` $: T^{\{x\}[x \mapsto \bot]} = T^{\bot}$ 😱

No Reachability-Generic Code!

- Substitution with the non-track qualifier must yield a set.
- Otherwise, reachability tracking can be subverted.
- Reachability polymorphism is imprecise, requires code duplication for track/non-track => impractical!

A NEW REACHABILITY MODEL IN $\lambda^{\blacklozenge}$

```
new Ref(42)                : Ref[Int]^{♦}

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

val i = 42                 : Int^∅

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

val z = !y                 : Int^∅

x := 0                     : Unit^∅
```

Intuition: Reachability Types & Qualifiers

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

$$\cancel{q \in \{\bot\} \uplus \mathscr{P}_{\mathsf{fin}}(\mathsf{Var})}$$

$$q \in \mathscr{P}_{\mathsf{fin}}(\mathsf{Var} \uplus \{\blacklozenge\})$$

Computation t yields a *T* value which <u>may</u> reach all variables in *q*.

~~$\bot$ is untracked (often omitted).~~

$\varnothing$ is untracked (often omitted).

~~$\varnothing$ means “fresh”, no sharing w. context.~~

$\blacklozenge$ means “contextually fresh”, can grow with unobserved future locations at run time.

A NEW REACHABILITY MODEL IN $\lambda^{\blacklozenge}$

Contextual Freshness:

$\sigma | \mathbf{new\ Ref}(42) : \mathtt{Ref[Int]}^{\{\blacklozenge\}} \longrightarrow \sigma, \ell = 42 | \ell : \mathtt{Ref[Int]}^{\{\ell\}}$

where $\ell \notin \mathrm{dom}(\sigma)$

A NEW REACHABILITY MODEL IN $\lambda^{\blacklozenge}$

```
def id(x: T{♦}): T{x} = x

def id(x: T∅): T{x} = x

val x: T{x,a,b} = …;    val y: T{y,z} = …;    val i: T∅ = …

id(x) // : T{x}[x ↦ {x,a,b}] = T{x,a,b}

id(y) // : T{x}[x ↦ {y,z}]   = T{y,z}

id(i) // : T{x}[x ↦ ∅]       = T∅
```

😀

```
def fakeid(x: T{♦}): T{x} = alloc()
```

Type error:

alloc(): $T^{\{\blacklozenge\}} \not<: T^{\{x\}}$

😀

EAGER VS. ON-DEMAND REACHABILITY

Typing Context in λ^* vs. $\lambda^\blacklozenge$

`val x = new Ref(42)`	$\texttt{x:Ref[Int]}^{\{\texttt{x}\}}$	$\texttt{x:Ref[Int]}^{\{\blacklozenge\}}$
`val y = x`	$\texttt{y:Ref[Int]}^{\{\texttt{y,x}\}}$	$\texttt{y:Ref[Int]}^{\{\texttt{x}\}}$
`val z = x`	$\texttt{z:Ref[Int]}^{\{\texttt{z,x}\}}$	$\texttt{z:Ref[Int]}^{\{\texttt{x}\}}$
`val w = z`	$\texttt{w:Ref[Int]}^{\{\texttt{w,z}\}}$	$\texttt{w:Ref[Int]}^{\{\texttt{z}\}}$

$$\frac{x : T^{q^*,x} \in \Gamma}{\Gamma \vdash x : T^{q^*,x}} \qquad \frac{x : T^{q} \in \Gamma}{\Gamma \vdash x : T^{x}}$$

ON-DEMAND REACHABILITY

Reachability Chains

```
x:Ref[Int]{♦}

y:Ref[Int]{x}

z:Ref[Int]{x}

w:Ref[Int]{z}
```

Examples:

```
{w} <: {z} <: {x}

{y} <: {x,y,z,w}

{w,y} <: {z,x} <: {x}

{x} <: {x,♦}
```

$\{x\} \not<: \{\blacklozenge\}$

Qualifier Subtyping (Excerpt):

$$\frac{p \subseteq q \subseteq \mathrm{dom}(\Gamma) \cup \{\blacklozenge\}}{\Gamma \vdash p <: q}$$

$$\frac{x : T^q \in \Gamma \quad \blacklozenge \notin q}{\Gamma \vdash \{x\} <: q}$$

$$\frac{\Gamma \vdash q <: r}{\Gamma \vdash p, q <: p, r}$$

$$\frac{\Gamma \vdash p <: q \quad \Gamma \vdash q <: r}{\Gamma \vdash p <: r}$$

ON-DEMAND REACHABILITY

More Precise Reachability Polymorphism

```
def foo(x: T{a}): T{a,x} = a:=!a+1; x  // (x: T{a} => T{a,x}){a}
```

Eager model demands reflexive transitive reachability assignment. a's only purpose here is specifying legal overlap, but we can't get rid of it!

```
def foo(x: T{a,♦}): T{x} = a:=!a+1; x  // (x: T{a,♦} => T{x}){a}
```

On-demand model preserves reachability chains, giving us what we really want.

ON-DEMAND REACHABILITY

When is Reflexive-Transitive Reachability Required?

```
val c1 = new Ref(0)                 // : Ref[Int]^{c1} ⊣ c1: Ref[Int]^♦
def f(x : Ref[Int]^♦) = !c1 + !x    // : (x: Ref[Int]^♦ => Int)^{c1}
val c2 = c1                         // : Ref[Int]^{c2}
f(c2)                               // : WRONG: {c1} ∩^♦ {c2} = ♦, accept
                                    // : RIGHT: {c1}* ∩^♦ {c2}* = {c1,♦} ≠ ♦, reject!
```

Separation/Overlap Checks Must be Eager!

- When applying functions/type abstractions.
- When composing effects in the quantale framework.

TYPE-AND-REACHABILITY ABSTRACTION

$\lambda^{\blacklozenge}$ Smoothly Scales to an $F^{\blacklozenge}_{<:}$- Calculus, Yay!

$$\texttt{// try}(\forall A^z <: \texttt{Top}^{\blacklozenge}.(\texttt{CanThrow}^{\blacklozenge} \Rightarrow A^z)^{\blacklozenge} \Rightarrow A^z)^{\varnothing}$$

$$\textbf{def}\ \texttt{try}[A^{\blacklozenge}](\texttt{block}: (\texttt{CanThrow}^{\blacklozenge} \Rightarrow A)^{\blacklozenge}): A$$

Observable Separation for Universal Types

- Track free variables, consistent with view as closure records.
- Just as function types, have self-qualifiers for scope transfers.
- To prevent interference from uncontrolled aliasing, type abstractions are separated from their arguments
- If full separation is too strict, we may adjust the universal type's domain's qualifier for degrees of overlap.

Pair Constructor Signature (Strictly Disjoint Components)

$$\texttt{pair}(\forall A^x <: \texttt{Top}^{\blacklozenge}.\forall B^y <: \texttt{Top}^{\blacklozenge}.((u:A^{\{x\}}, v:B^{\{y\}}) \Rightarrow (A^{\{u\}}, B^{\{v\}})^{\{x,y\}})$$

Version With Overlapping Reachability

$$\texttt{pair}(\forall A^x <: \texttt{Top}^{\blacklozenge}.\forall B^y <: \texttt{Top}^{\{\blacklozenge,x\}}.((u:A^{\{x\}}, v:B^{\{y\}}) \Rightarrow (A^{\{u\}}, B^{\{v\}})^{\{x,y\}})$$

DATA TYPES

```
type Pair[A^a <: Top♦, B^b <: Top♦] = ([C^c <: Top♦] => (((A^a, B^b) => C^c)♦ => C^c)^{a,b})^{a,b}

// shorthand (implicit qualifiers): ([C♦] => (((A, B) => C)♦ => C)^{a,b})^{a,b}
def Pair[A♦, B♦](a: A, b: B): Pair[A, B] = [C♦] => (f: (A, B) => C) => f(a, b)

def fst[A♦, B♦](p: Pair[A, B]): A = p((a, b) => a)
def snd[A♦, B♦](p: Pair[A, B]): B = p((a, b) => b)
```

Implicit Capability-Polymorphic Elimination!

```
p { (a, b) =>
  file.write(a);
  file.write(b)
}
```

```
try { throw =>
  p { (a, b) =>
    if (a == 0) throw("error")
    file.write(a);
    file.write(b)
}}
```

Encoding Data Types

- Can build on standard System F encodings of data types modulo reachability sets. Cf., e.g.,
  - Corrado Boehm and Alessandro Berarducci: Automatic Synthesis of Typed Lambda-Programs on Term Algebras. Theoretical Computer Science, 1985
  - https://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html
  - https://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt

DATA TYPES

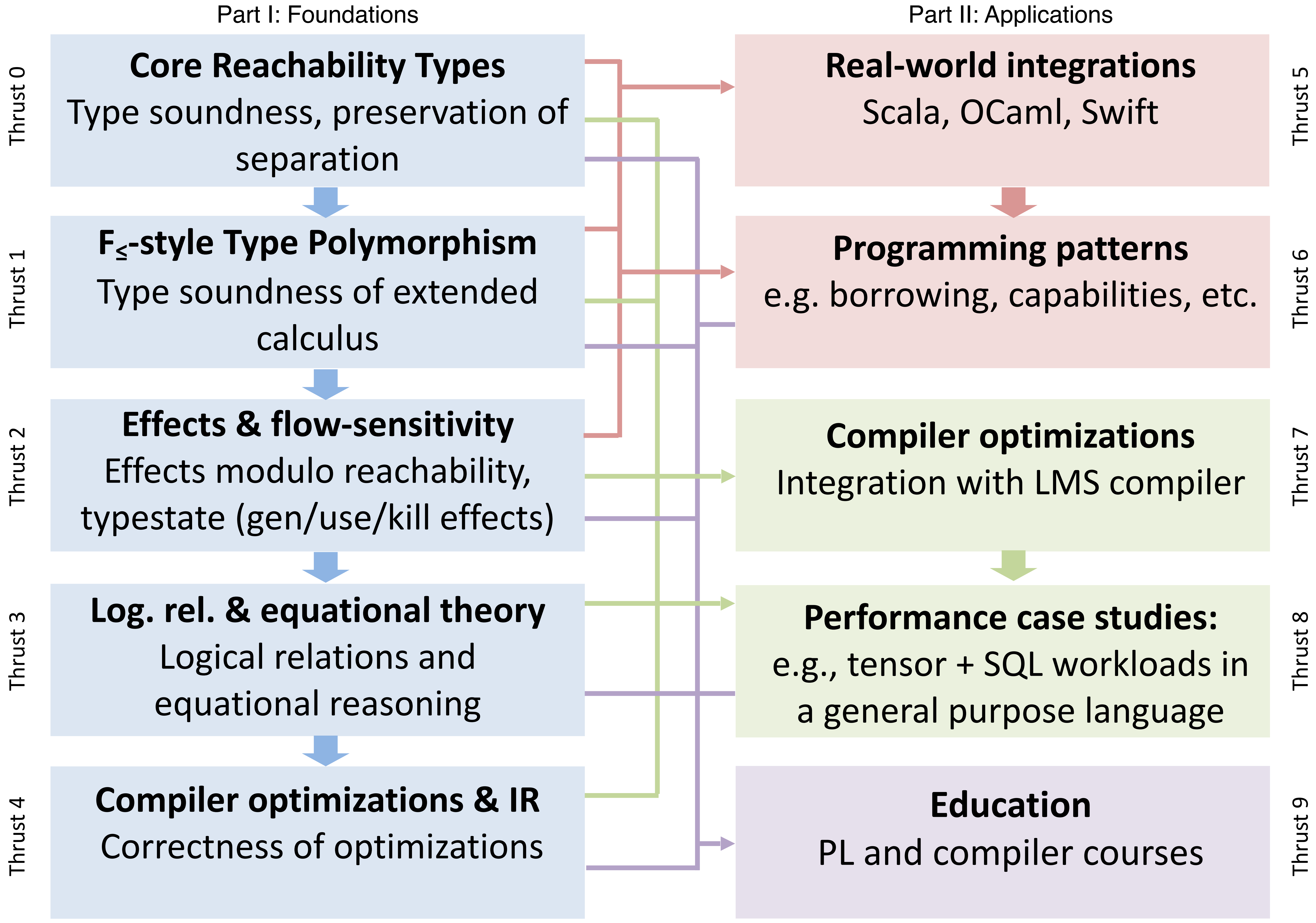
Revisiting the Counter Example

```
type {T} = () => T               // thunk type
// counter: Int => μp.Pair[{Unit}^{p},{Unit}^{p}]^∅
def counter(n: Int) = {
  val c = new Ref(n)             // : Ref[Int]^{c}
  (() => c += 1, () => c -= 1)  // : Pair[{Unit}^{c},{Unit}^{c}]^{c}
}
```

```
// replace p with bound name ctr:
val ctr = counter(0) // : Pair[{Unit}^{ctr},{Unit}^{ctr}]^{ctr}
// captured variables abstracted by name ctr:
val incr = fst(ctr)  // : {Unit}^{ctr}
val decr = snd(ctr)  // : {Unit}^{ctr}
```

REACHABILITY TYPES

Research Roadmap, Artifacts @ `https://github.com/TiarkRompf/reachability`



- **Aim: Provide end-to-end mechanized Coq proofs for Thrusts 0-4!**

- Core system (Thrust 0, 6): OOPSLA 2021

  Reachability Types: Tracking Aliasing and Separation in Higher-Order Functional Programs

- Generics and data types (Thrust 1, 6): Mechanization completed, Under submission.

  Polymorphic Reachability Types: Tracking Aliasing and Separation in Higher-Order Generic Programs

- Compiler IR (Thrust 3, 4, 7, 8): Prototype in progress. Under submission.

  Graph IRs for Impure Higher-Order Languages
  Making Aggressive Optimizations Affordable with Precise Effect Dependencies

“We both know what memories can bring

They bring diamonds and rust”

— Joan Baez (1975)

REACHABILITY IMPLIES ALIASING

(But Not Vice Versa!)

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

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

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

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

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

val v = (() => x)() : Ref[Int]^{v,x}
```

- Reachability sets in context are immutable once introduced!
- Cheaper to compute and maintain than full aliasing.
- Reachability is sufficient to check if two expressions share aliasing, e.g.,
  - Qualifiers of `y` and `w` overlap.
  - Qualifiers of `u` and `x` are disjoint.
- **Reachability + separation is sufficient to model most uses of Rust-like systems!**