Free to Move: Reachability Types with Flow-Sensitive Effects for Safe Deallocation and Ownership Transfer

HAOTIAN DENG, Purdue University, USA
SIYUAN HE, Purdue University, USA
SONGLIN JIA, Purdue University, USA
YUYAN BAO, Augusta University, USA
TIARK ROMPF, Purdue University, USA

We present a flow-sensitive effect system for reachability types that supports explicit memory management, including Rust-style move semantics, in higher-order impure functional languages. Our system refines the existing reachability qualifier with polymorphic *use* and *kill* effects that record how references are read, written, transferred, and deallocated. The effect discipline tracks operations performed on each resource using qualifiers, enabling the type system to express ownership transfer, contextual freshness, and destructive updates without regions or linearity. We formalize the calculus, its typing and effect rules, and a compositional operational semantics that validates use-after-free safety. All metatheoretic results, including preservation, progress, and effect soundness, are mechanized. The system models idioms such as reference deallocation, move semantics, reference swapping, while exposing precise safety guarantee. Together, these contributions integrate reachability-based reasoning with explicit resource control, advancing the state of the art in safe manual memory management for higher-order functional languages.

1 Introduction

In the past decades, there has been a rich body of work studying regions that established how static analyses can recover predictable memory lifetimes [Aiken et al. 1995; Lippmeier 2013; Tofte and Talpin 1994], while substructural and uniqueness type systems explored disciplined resource use in higher-order settings [Bernardy et al. 2018; Lorenzen et al. 2023; Marshall et al. 2022; Milano et al. 2022]. These lines of work underpin the design of modern languages such as Rust, whose success demonstrates the practical value of static ownership tracking for systems code [Clarke et al. 1998; Noble et al. 2023].

Within this landscape, reachability types (RT) offer a complementary perspective by describing how references flow through higher-order impure functional programs [Bao et al. 2021; Wei et al. 2024]. Recent extensions enrich this theory with cyclic references [Deng et al. 2025], as well as lexical memory management and region-style reasoning [He et al. 2025]. RT thereby capture sharing, separation, and lifetimes, suggesting a path toward integrating explicit memory control with expressive higher-order functions.

This paper takes that path by developing an *explicit* memory management discipline for reachability types. We introduce flow-sensitive *kill* effects on top of *use* effects that formalizes an effect system on top of RT, enabling precise tracking of when references may be dereferenced, moved, or deallocated. The resulting system models regions and lexical memory management via qualifiers, supports higher-order impure programs, and rules out use-after-free errors while preserving the expressiveness needed for ownership transfer and manual resource control.

The rest of the paper develops the design, metatheory, and applications of this effect discipline. We begin with motivating examples that illustrate how qualified reachability and effects interact to capture common idioms such as ownership transfer and swap operations. We then present the

Authors' Contact Information: Haotian Deng, Purdue University, West Lafayette, USA, deng254@purdue.edu; Siyuan He, Purdue University, West Lafayette, USA, he662@purdue.edu; Songlin Jia, Purdue University, West Lafayette, USA, jia137@purdue.edu; Yuyan Bao, Augusta University, Augusta, USA, yubao@augusta.edu; Tiark Rompf, Purdue University, West Lafayette, USA, tiark@purdue.edu.

core calculus, including its syntax, typing rules, and operational semantics, together with a notion of flow-sensitive effects. A mechanized soundness proof establishes preservation and progress, ensuring that well-typed programs respect the intended safety guarantees, including, specifically, safety of deallocation and move semantics. Finally, we discuss related works and how our approach connects to existing approaches in the literature.

Contributions.

- **A polymorphic, flow-sensitive effect system.** We design and formalize a qualified effect system that supports polymorphism over both types and flow-sensitive effects, enabling precise tracking of reachability of resources in programs with side effects.
- ***Use* effect—read and write operations.** We refine effect tracking with a qualified *use* effect that precisely accounts for dereferencing and assignment along reachability, explicitly separating read and write operations on references from mere mention.
- ***Kill* effect—destructive operations.** We introduce a destructive *kill* effect, and a corresponding **free** construct that explicitly frees memory and invalidates any subsequent dereference or assignment.
- **Ownership transfer.** We capture Rust-style ownership transfer using a sequence of a *use* effect followed by a *kill* effect, along with a **move** construct. The moved resource becomes unusable and rebound under a fresh name, enabling uniqueness without scoped regions, linear types, or imposing global invariants.
- **Static effect safety guarantees.** We prove soundness (progress and preservation) of deallocation and move semantics via a flow-sensitive effect system, ruling out use-after-free sequences; all results are mechanized.

Summary. In summary, we demonstrate how to achieve safe deallocation, Rust-style move semantics, ownership transfer, and lifetime guarantees in higher-order impure functional languages such as Scala and OCaml.

2 Reachability Types (RT)

In this section we first revisit the key components of reachability types as introduced by Wei et al. [2024], focusing on how reachability qualifiers, contextual freshness, and reference types interact with one another (Section 2.1). We then extend this base system with a flow-sensitive effect layer that captures reads, writes, deallocation, and ownership transfer (Section 2.2). We formalize those ideas in the $\mathsf{F}^{\blacklozenge}_{\varepsilon<:}$ calculus in Section 3.

2.1 Key Ideas of RT

2.1.1 Reachability Qualifiers: Tracking Reachable Resources in Types. Types in RT are of the form $T^{\,p}$, where p is a *reachability qualifier*, indicating the set of variables and locations that may be reached from the result of an expression. Reachability qualifiers may optionally include the freshness marker ♦, indicating a fresh, unnamed resource. In the following example, evaluating the expression `new Ref(0)` results in a *fresh* value: it is not yet bound to a name, but must be tracked. The typing context `[ counter: Ref[Int]♦ ]` following a reverse turnstile "⊣" means `counter` reaches a fresh value:

```
val counter = new Ref(0)        // : Ref[Int]^counter ⊣ [ counter: Ref[Int]♦ ]
```

RT keep reachability sets minimal, *e.g.*, variable `counter` tracks exactly itself. When an alias is created for variable `counter` as shown below, RT assign the one-step reachability set `counter2` to variable `counter2`.

```
val counter2 = counter          // : Ref[Int]^counter2 ⊣ [ counter2: Ref[Int]^counter, counter: Ref[Int]♦ ]
```

We can retrieve the complete reachability set by computing its transitive closure with respect to the typing context [Wei et al. 2024].

Functions also track reachability: their reachability qualifier includes all *captured variables*. In the following example, function `inc` captures the free variable `counter`:

```
def inc(n: Int) = { counter := !counter + n } // : inc: (Int => Unit)^counter ⊣ [ counter: Ref[Int]♦ ]
```

2.1.2 Freshness Marker in Function Arguments: Contextual Freshness. The presence of the freshness marker ♦ in a function argument's qualifier indicates that the argument may only reach *unobservable* resources, meaning that its reachable locations must remain separate from those of the function. Thus, function applications must satisfy the *separation constraint*, requiring that the argument's reachability qualifier is disjoint from that of the function.

```
def id(x: T♦): T^x = x                                   // : ((x: T♦) => T^{x})^∅
```

The type means that function `id` cannot capture anything from its context, and it accepts arguments that may reach unobservable resources. A function application that violates this separation constraint results in a type error:

```
def update(x: Ref[Int]♦): Unit = { counter := !(x) + 1 }  // : ((x: Ref[Int]♦) => Unit)^counter
update(counter) // Error! variable counter overlaps with function update
```

The above function application violates the separation constraint: the passing argument `counter` overlaps with function `update`.

2.1.3 Reference Type: Mutable Cells. So far, the reference type examples have only used primitive referent types (*e.g.* `Int`) as the referent type. Since these have untracked qualifiers (∅)[1], the qualifiers are elided. Wei et al. [2024]'s system supports reference types with tracked referent qualifiers, enforcing that an assigned referent must match the exact specified qualifier:

```
val a = ...              // : T^a
val b = ...              // : T^b
val cell = new Ref(...) // : Ref[T^a]⋯
cell := a // Okay
cell := b // Error! Referent qualifier mismatch!
```

```
val a = ...              // : T^a
val b = ...              // : T^b
val cell = new Ref(...) // : Ref[T^{a,b}]⋯
cell := a // Okay, T^a <: T^{a,b}
cell := b // Okay, T^b <: T^{a,b}
```

As shown above (left), since reference `cell` has `a` as its referent qualifier, it is only permitted to be assigned a value with qualifier `a`. Assigning it with a different qualifier, *e.g.*, `b`, results in a type error. On the other hand, if `cell` is created with a widened referent qualifier `a,b`, as shown above (right), then both assignments are allowed, since both `T$^{\trackvar{a}}$` and `T$^{\trackvar{b}}$` are subtypes of `T$^{\trackvar{a,b}}$`.

2.2 Extending RT with Effects

2.2.1 Effects: Tracking Reads, Writes, Deallocation, and Ownership Transfer. In fact, without flow-sensitive effects, RT can already model a rich set of resource usage patterns, using regions to model lexically scoped lifetimes [He et al. 2025]. However, regions are not expressive enough to model patterns that involve non-lexical lifetimes, such as explicit deallocation and ownership transfer. To track such patterns, we extend RT with a flow-sensitive effect system that records how tracked resources are manipulated as the program executes. Intuitively, we distinguish two families of effects: a *use* effect marks that a resource is observed or mutated, whereas a *kill* effect records that a resource becomes unavailable for any subsequent use. The rest of this section illustrates how these informal concepts surface in the formal typing judgment.

[1]The untracked qualifier (∅) indicates that a value has no reachable locations. Primitive values are usually untracked, since they represent pure, location-independent data.

This extension follows the framework of Deng et al. [2025], but tailors the effect alphabet to the operations that appear in our motivating examples. We write effects in record syntax $\{\mathcal{u} : \alpha; \not\mathcal{k} : \beta\}$ where α and β are finite sets of variables annotated with labels $\mathcal{u}$ (*used* variables) and $\not\mathcal{k}$ (*killed* variables). Empty record fields are omitted for readability (see Figure 1). Function types carry both reachability qualifiers and effects, and the typing rules sequence effects when composing computations. This combination gives us a uniform view: qualifiers continue to describe what resources a value can reach, while effects describe how those resources are accessed as the program executes.

2.2.2 Use Effects for Reads and Writes. Whenever an effectful operation touches a tracked resource, the effect system records that access. Reads and writes therefore induce use effects on the resource's reachability set. In the following snippet the assignment does not just update the store; it also registers a $\mathcal{u}$ effect that must be accounted for at the call site:

Invoking an effectful operation (*e.g.*., reference assignment) on a tracked resource induces a use effect on its reachability set:

```
val cell = new Ref(elm) // : Ref[T^elm]^cell
cell := ... // : {u : cell} ← using cell but not using elm
```

2.2.3 Kill Effects for Deallocation. Explicit deallocation is expressed via kill effects. Killing a resource removes it from the set of usable names, and the static effect sequencing ensures that no later computation can accidentally rely on a dead value. The effect trace $\{\not\mathcal{k} : \texttt{cell}\}$ witnesses that the subsequent computation must stop using the reference after the deallocation point:

Deallocating a tracked resource induces a kill effect on its reachability set, prohibiting further uses in the program:

```
free(cell) // : {k : cell} ← killing cell
```

2.2.4 Main Invariant: No Use-After-Kill. The cumulative effect discipline ultimately enforces a global invariant on program traces: the main invariant we enforce is *no use-after-kill*: once a variable is killed, it cannot be used again. This invariant is enforced through the type system, which tracks effects in function types and checks effect sequencing at function application sites.

```
!cell // : Error! Use of killed variable cell!
```

2.2.5 Use vs. Mention. To keep the effect system permissive for read-only observations, we separate *using* a resource from merely mentioning it in a type. Mentions do not induce an effect, so expressions mentioning resources remain pure. Moreover, merely *mentioning* a killed variable remains pure and is allowed:

```
def size(c: Ref[T^elm]♦) = 0 // : (c: Ref[T^elm]⋯) => Int^∅ ∅
size(cell) // : Int^∅ ∅ ← ok: mere mention of cell
```

2.2.6 Precise Effect Tracking. The interaction between reachability and effects is subtle: effect annotations should overapproximate the resources that an operation actually manipulates. With one-step store reachability proposed in Deng et al. [2025], where reachability qualifiers only track individual memory locations instead of transitively reachable locations through the store, it is suitable to implement an effect extension on top of it to precisely track used and deallocated memory locations: reassigning `cell` would mark `cell` as *used* without incorrectly propagating the use effect to `elm`:

```
cell := ...          // : {u : cell}              ← using cell but not using elm
free(cell)           // : {u : cell} ▷ {k : cell} ← killing cell
use(elm)             // : Unit^∅ {u : elm}        ← using elm, okay
```

2.2.7 Idempotent Kill Effects. One property of kill effects is that they are *idempotent*: killing a resource multiple times is allowed, and has the same effect as killing it once:

```
free(cell)  // : {𝓀 : cell} ⟵ killing cell
free(cell)  // : {𝓀 : cell} ⟵ killing cell again, okay
!cell       // : Error! Use of killed variable cell!
```

We describe the idempotent semantics of deallocation in **??**.

2.2.8 Move Effect: Explicit, Rust-style Move Semantics. With move effects, we explicitly model Rust-style ownership transfer that disables further access to the moved variable. The **move** construct induces a *move effect* on reference cell `r`, transferring its ownership to `s`.

```
val r = new Ref(0)    // : Ref[Int]^r
val s = move r        // : {𝓊 : r; 𝓀 : r} ⟵ transfer ownership from r to s; r becomes unusable
```

After the move, `r` is unusable, and any further use of `r` is a type error:

```
r := 5                // : Error! Cannot use moved resource r
```

However, `s` now owns the resource originally owned by `r`, and can be used safely:

```
s := 41               // : {𝓊 : s}
free(s)               // : {𝓊 : s} ▷ {𝓀 : s}
```

2.2.9 Borrow + Selective CPS (Shift/Reset) Equivalence. The effect calculus also interacts smoothly with control abstractions that re-express ownership transfer without primitive moves. The ownership transfer below uses an explicit `move`:

```
// Direct-style move
val r = new Ref(0)    // : Ref[Int]^r
val s = move r        // : {𝓊 : r; 𝓀 : r} ⟵ transfer ownership from r to s; r becomes unusable
```

However, a primitive `move` is not essential: the same ownership transfer can be expressed by a `borrow` combinator that introduces a fresh, separated handle and prevents the surrounding continuation from mentioning the old name. This can be accomplished with the following borrow combinator:

```
def borrow[A♦,B♦](x: A♦) (block: (A♦ => B♦)^∅): B♦ = block(x)
```

With selective CPS transformation, `move` is equivalent to a continuation using the `borrow` combinator (below on the left), and can also be expressed using `shift`/`reset` (below on the right).

```
// Equivalent 1: selective CPS transformation
val r = new Ref(0)        // : Ref[Int]^r
def k(s: Ref[Int]^s): Ref[Int]^s = s
val s = borrow(r){ z => k(z) }
```

```
// Equivalent 2: shift/reset
val r = new Ref(0)        // : Ref[Int]^r
val s = reset { shift k {
  borrow(r){ z => k(z) } }}
```

We retain a direct-style **move** operator to surface the destructive effect without relying on additional control operators or CPS transformations.

3 $F^{♦}_{\varepsilon<:}$: Reachability Types with Destructive Effects

In this section, we formally present $F^{♦}_{\varepsilon<:}$. Section 3.1 introduces variables, terms, ordinary and qualified types, effects, and environments; Section 3.2 formalizes the typing/effect judgments for expressions; Section 3.3 specifies subtyping for types, qualifiers, and effects; Section 3.4 presents the small-step operational semantics, particularly highlighting the treatment of store deallocation and ownership transfer; and Section 3.5 includes key lemmas such as type safety (progress and preservation), which implies effect safety.

Syntax $\mathsf{F}^{\blacklozenge}_{\varepsilon<:}$

x, y, z	$\in$	Var	Variables
f, g, h	$\in$	Var	Function Variables
X	$\in$	Var	Type Variables
S, T, U, V	$::=$	$\mathsf{Unit} \mid f(x : Q) \to R \mid \mathsf{Ref}\ [Q]$ $\mid \top \mid X \mid \forall f(X^x <: Q).Q$	Types
B	$::=$	Unit	Base Types
t, t_1, t_2	$::=$	$c \mid x \mid \lambda f(x).t \mid t_1\ t_2 \mid \mathbf{ref}\ t \mid !\ t \mid t_1 := t_2$ $\mid \Lambda f(X^x).t \mid t\ [Q] \mid \mathbf{free}\ t \mid \mathbf{move}\ t$	Terms
p, q, r, w	$\in$	$\mathcal{P}_{\mathsf{fin}}(\mathsf{Var} \uplus \{\blacklozenge\})$	Type Qualifiers
α, β, γ	$\in$	$\mathcal{P}_{\mathsf{fin}}(\mathsf{Var})$	Effect Qualifiers
O, P, Q, R	$::=$	T^q	Qualified Types
$\varepsilon, \varepsilon_1, \varepsilon_2, \varepsilon_3$	$::=$	$\{\mathcal{u} : \alpha; \mathcal{k} : \alpha\}$	Effects
φ	$\in$	$\mathcal{P}_{\mathsf{fin}}(\mathsf{Var})$	Observations
Γ	$::=$	$\varnothing \mid \Gamma, x : Q \mid \Gamma, X^x <: Q$	Typing Environments

Type, Qualifier, and Effect Notations

$\varnothing$	$:=$	$\{\mathcal{u} : \varnothing; \mathcal{k} : \varnothing\}$	Pure Effect
p, q	$:=$	$p \cup q$	Qualifier Union
x	$:=$	$\{x\}$	Single Variable Qualifier
$\blacklozenge$	$:=$	$\{\blacklozenge\}$	Single Fresh Qualifier

Fig. 1. The syntax of $\mathsf{F}^{\blacklozenge}_{\varepsilon<:}$.

3.1 Syntax

Figure 1 presents the surface syntax for $\mathsf{F}^{\blacklozenge}_{\varepsilon<:}$. It extends Deng et al. [2025]'s reachability types system with constructs for explicit deallocation and ownership transfer, along with a flow-sensitive effect system.

3.1.1 Qualifiers. Type qualifiers $p, q, r, w \in \mathcal{P}_{\mathsf{fin}}(\mathsf{Var} \uplus \{\blacklozenge\})$ are finite sets of names, optionally including the distinct freshness marker $\blacklozenge$. Effect qualifiers α, β, γ range over the same carrier set but do not include the fresh marker $\blacklozenge$ (See Section 3.2.6). Effect qualifiers are parameterized components of an effect. For readability we often write qualifiers as comma-separated lists of names rather than set brackets (See **??**).

3.1.2 Effects. We use $\varepsilon, \varepsilon_1, \varepsilon_2, \varepsilon_3$ to represent effect variables. Effects are records carrying two qualifiers, representing its *use* component ($\mathcal{u}$) and *kill* component ($\mathcal{k}$). Effect constructor $\{\mathcal{u} : \alpha; \mathcal{k} : \alpha\}$ is parameterized by the *use* and the *kill* component respectively. When one component is empty, we may omit the empty component by writing $\{\mathcal{u} : \alpha\}$ or $\{\mathcal{k} : \alpha\}$.

3.1.3 Types and Qualified Types. We separate ordinary types T from qualified types Q. A qualified type has the form $Q \equiv T^q$, where the qualifier q tracks a finite set of names relevant to T (e.g., potential aliases/reachable variables). We use S, T, U, V to range over ordinary types and O, P, Q, R over qualified types.

Base Types. Base types do not carry qualifiers as a sub-component, and in our system, they are Unit, the unit type, and $\top$, the top type.

Dependent Function Types. Dependent Function types are of the form $f(x : T^{q}) \to^{\varepsilon} S^{p}$, where the codomain S^{p} may depend on the argument x and the self-reference f. Function types also carry a latent effect ε. The latent effect may depend on both the function self-reference f and argument x, just as the return type-and-qualifier S^{p}.

Reference Types. Reference types have the form Ref $[T^{q}]$, where T is the referent type and q is the referent qualifier that tracks the reachable variables through dereferencing.

Universal Types. Universal types $\forall\, f(X^{x} <: Q).\, Q$ quantify over a type variable X with a qualifier x under a qualified upper bound Q; the self reference f is brought into scope so that both types and qualifiers in the body may mention the universal type as a whole.

Terms. Constants c inhabit base types $B ::= \mathsf{Unit}$. $\lambda f(x).t$ is a recursive function (binding self f and parameter x in t); application has the form of $t_1\ t_2$. Additionally, we have reference manipulating terms **ref** t, $!\,t$, and $t_1 := t_2$, which perform reference allocation, dereference, and assignment respectively. $\Lambda f(X^{x}).t$ is a bounded type abstraction over a type variable X with qualifier x, $t\ [Q]$ instantiates with a qualified type argument Q. The forms **free** t and **move** t are primitive effectful operations whose effects are tracked in the static semantics.

3.2 Static Typing

Term typing in $\mathsf{F}^{\blacklozenge}_{\varepsilon<:}$ builds on Deng et al. [2025]; Wei et al. [2024]. We write the judgment as $\Gamma^{\varphi} \vdash t : T^{q}\ \varepsilon$ (see Figure 2). Under observation filter φ, the term t evaluates to a value of type T with reachability qualifier q and the evaluation may incur effect ε. The qualifier q constrains which parts of the environment t may reach. The effect component ε tracks side effects only on entities visible through φ and composes sequentially in subsequent derivations. It is also one-step by default, and its transitive closure through the context is only computed upon effect composition (See Figure 5). Consequently, the judgment is precise about both the reachable results (via q) and the incurred effects (via ε).

Figure 2 presents a representative subset of typing rules that incorporate effects into the polymorphic setting. We now describe the typing rules with a particular focus on how reachability types integrate with effects.

3.2.1 Variables and Constants. T-VAR is pure: reading a variable produces no effects and the result qualifier precisely names the accessed path. T-CST is also pure; a constant is untracked and cannot reach any location on the heap (qualifier $\varnothing$).

3.2.2 Reference Introduction. T-REF allocates a reference but does not add effects beyond those already incurred when evaluating the argument; allocation is tracked only by the result qualifier, which becomes $\blacklozenge$. As in Wei et al. [2024], we do not allow the referent qualifier to be fresh.

3.2.3 Use Operations on References. T-ASSGN is the standard reference assignment rule, which induces a use effect on the written reference. Intuitively, the effects observed are a result of first evaluating the reference and then the assigned value, composed as $\varepsilon_1 \triangleright \varepsilon_2$, finally followed by a single write on the reference qualifier p. The assignment rule does not induce a kill effect, as assignment does not disable the use of the underlying reference. T-DEREF reveals the referent's qualifier and registers exactly one use effect on the dereferenced reference.

3.2.4 Reference Deallocation and Move. T-FREE and T-MOVE are the only rules that induce a destructive *kill* effect.

Both first realize any prior effects of their operand, then induces the destructive *kill* effect. Note that **move** first induces a *use* effect on the reference being moved. This design choice rules out

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

$$\frac{y : T^{q} \in \Gamma \quad y \in \varphi}{\Gamma^{\varphi} \vdash y : T^{y}\ \varnothing}\ (\textsc{t-var}) \qquad \frac{c \in B}{\Gamma^{\varphi} \vdash c : B^{\varnothing}\ \varnothing}\ (\textsc{t-cst}) \qquad \frac{\Gamma^{\varphi} \vdash t : Q\ \varepsilon_1 \quad q, \varepsilon_2 \subseteq \varphi, \blacklozenge \quad \Gamma \vdash Q\ \varepsilon_1 <: T^{q}\ \varepsilon_2}{\Gamma^{\varphi} \vdash t : T^{q}\ \varepsilon_2}\ (\textsc{t-sub})$$

$$\frac{\Gamma^{\varphi} \vdash t : T^{q}\ \varepsilon \quad \blacklozenge \notin q}{\Gamma^{\varphi} \vdash \mathbf{ref}\ t : (\mathrm{Ref}\ T^{q})^{\blacklozenge}\ \varepsilon}\ (\textsc{t-ref}) \qquad \frac{\Gamma^{\varphi} \vdash t : \mathrm{Ref}\ [T^{q}]^{p}\ \varepsilon\ q \subseteq \varphi}{\Gamma^{\varphi} \vdash\ !t : T^{q} \quad \varepsilon \rhd \{u : p \setminus \blacklozenge\}}\ (\textsc{t-deref}) \qquad \frac{\Gamma^{\varphi} \vdash t_1 : \mathrm{Ref}\ [T^{q}]^{p}\ \varepsilon_1\ \Gamma^{\varphi} \vdash t_2 : T^{q}\ \varepsilon_2}{\Gamma^{\varphi} \vdash t_1 \coloneqq t_2 : \mathrm{Unit}^{\varnothing}\ \varepsilon_1 \rhd \varepsilon_2 \rhd \{u : p \setminus \blacklozenge\}}\ (\textsc{t-assgn})$$

$$\frac{(\Gamma,\ f : F,\ x : P)^{q,x,f} \vdash t : Q\ \varepsilon \quad F = (f(x : P) \to^{\varepsilon} Q)^{q} \quad q \subseteq \varphi}{\Gamma^{\varphi} \vdash \lambda f(x).t : F\ \varnothing}\ (\textsc{t-abs}) \qquad \frac{(\Gamma,\ f : F,\ X^{x} <: P)^{q,x,f} \vdash t : Q\ \varepsilon \quad F = (\forall f(X^{x} <: P).^{\varepsilon} Q)^{q} \quad q \subseteq \varphi}{\Gamma^{\varphi} \vdash \Lambda f(X^{x}).t : F\ \varnothing}\ (\textsc{t-tabs})$$

$$\frac{\Gamma^{\varphi} \vdash t_1 : (f(x : T^{p}) \to^{\varepsilon_3} U^{r})^{q}\ \varepsilon_1 \quad \Gamma^{\varphi} \vdash t_2 : T^{p}\ \varepsilon_2 \quad \blacklozenge \notin p \quad \blacklozenge \in q \Rightarrow f \notin \hbar(\varepsilon_3) \cap r \quad r \subseteq \blacklozenge, \varphi, x, f}{\Gamma^{\varphi} \vdash t_1\ t_2 : (U^{r}\ \varepsilon_1 \rhd \varepsilon_2 \rhd \varepsilon_3)[p/x, q/f]}\ (\textsc{t-app})$$

$$\frac{\Gamma^{\varphi} \vdash t_1 : (f(x : T^{p \curlywedge q}) \to^{\varepsilon_3} U^{r})^{q}\ \varepsilon_1 \quad \Gamma^{\varphi} \vdash t_2 : T^{p}\ \varepsilon_2 \quad \blacklozenge \in p \Rightarrow x \notin (\mathrm{fv}(U), \hbar(\varepsilon_3) \cap r) \quad \blacklozenge \in q \Rightarrow f \notin (\mathrm{fv}(U), \hbar(\varepsilon_3) \cap r) \quad r \subseteq \blacklozenge, \varphi, x, f}{\Gamma^{\varphi} \vdash t_1\ t_2 : ((U^{r})\ \varepsilon_1 \rhd \varepsilon_2 \rhd \varepsilon_3)[p/x, q/f]}\ (\textsc{t-app-fresh})$$

$$\frac{\Gamma^{\varphi} \vdash t : (\forall f(X^{x} <: T^{p}).^{\varepsilon_3} Q)^{q}\ \varepsilon_1 \quad \blacklozenge \notin p \quad f \notin \mathrm{fv}(U) \quad p \subseteq \varphi \quad r \subseteq \blacklozenge, \varphi, x, f \quad Q = U^{r}}{\Gamma^{\varphi} \vdash t[T^{p}] : (Q\ \varepsilon_1 \rhd \varepsilon_3)[T^{p}/X^{x}, q/f]}\ (\textsc{t-tapp})$$

$$\frac{\Gamma^{\varphi} \vdash t : (\forall f(X^{x} <: T^{p \curlywedge q}).^{\varepsilon_3} Q)^{q}\ \varepsilon_1 \quad \blacklozenge \in p \Rightarrow x \notin \mathrm{fv}(U) \quad f \notin \mathrm{fv}(U) \quad p \subseteq \varphi \quad r \subseteq \blacklozenge, \varphi, x, f \quad Q = U^{r}}{\Gamma^{\varphi} \vdash t[T^{p}] : (Q\ \varepsilon_1 \rhd \varepsilon_3)[T^{p}/X^{x}, q/f]}\ (\textsc{t-tapp-fresh})$$

$$\frac{\Gamma^{\varphi} \vdash t : \mathrm{Ref}\ [T^{q}]^{p}\ \varepsilon}{\Gamma^{\varphi} \vdash \mathbf{free}\ t : \mathrm{Unit}^{\varnothing}\ \varepsilon \rhd \{\hbar : p \setminus \blacklozenge\}}\ (\textsc{t-free}) \qquad \frac{\Gamma^{\varphi} \vdash t : \mathrm{Ref}\ [T^{q}]^{p}\ \varepsilon}{\Gamma^{\varphi} \vdash \mathbf{move}\ t : \mathrm{Ref}\ [T^{q \setminus \blacklozenge}]^{\blacklozenge}\ \varepsilon \rhd \{u : p \setminus \blacklozenge; \hbar : p \setminus \blacklozenge\}}\ (\textsc{t-move})$$

Fig. 2. Selected Typing rules of $\mathsf{F}^{\blacklozenge}_{\varepsilon<:}$. We omit cyclic reference types and dual-component reference types from Deng et al. [2025] for simplicity. New constructs and additional typing constraints from Deng et al. [2025] are highlighted.

illegal "move-after-move" and "move-after-kill" sequences when the effect sequencing invariant is enforced (See Figure 5), without needing to introduce a separate "move" effect.

Another difference between the two rules is in their return types: T-FREE returns unit type, while T-MOVE returns the same payload type T, but with a fresh qualifier $\blacklozenge$, exposing the old reference with a unique access via a freshly allocated reference (See Section 3.4).

3.2.5 Abstractions. T-ABS is the introduction rule for abstractions: creating the abstraction does not touch the store; any heap effects are those declared as the *latent effect* of the abstraction and occur only when the function is applied. T-TABS is likewise pure for type abstraction; the polymorphic body may later perform effects when used, but none are incurred at the point of abstraction.

3.2.6 Application Rules. T-APP sequentializes effects in left-to-right evaluation order: first the callee ε_1, then the argument ε_2, then the latent effects declared on the abstraction ε_3. If the latent effect refers to the function self reference f or the parameter x, they are substituted with the actual argument qualifiers q and the self reference p respectively.

T-APP-FRESH is similar to T-APP, but allows the argument qualifier to contain $\blacklozenge$. It adds an additional condition that *a fresh argument must not be both returned and killed by the abstraction.* This rules out the possibility of any "dangling pointer" returned from the function application that becomes untracked by our system.

Subtyping $\boxed{\Gamma \vdash q <: q}\ \boxed{\Gamma \vdash T <: T}\ \boxed{\Gamma \vdash Q <: Q}\ \boxed{\Gamma \vdash Q\,\varepsilon <: Q\,\varepsilon}$

$$\frac{}{\Gamma \vdash B <: B}\ (\textsc{s-base}) \qquad \frac{\Gamma \vdash S <: T \quad \Gamma \vdash T <: S \quad q \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathsf{Ref}\ [S^{p}]^{q} <: \mathsf{Ref}\ [T^{p}]^{q}}\ (\textsc{s-ref})$$

$$\frac{\Gamma \vdash T <: S \quad \Gamma \vdash S <: U}{\Gamma \vdash T <: U}\ (\textsc{s-trans}) \qquad \frac{\Gamma \vdash P <: O \quad \Gamma, f : (f(x:O) \to^{\varepsilon_1} Q)^{\blacklozenge}, x : P \vdash Q\,\varepsilon_1 <: R\,\varepsilon_2}{\Gamma \vdash f(x:O) \to^{\varepsilon_1} Q <: f(x:P) \to^{\varepsilon_2} R}\ (\textsc{s-fun})$$

$$\frac{p \subseteq q \subseteq \blacklozenge, dom(\Gamma)}{\Gamma \vdash p <: q}\ (\textsc{q-sub}) \qquad \frac{f : T^{q} \in \Gamma \quad \blacklozenge \notin q}{\Gamma \vdash q, f <: f}\ (\textsc{q-self}) \qquad \frac{X^{x} <: T^{q} \in \Gamma \quad \blacklozenge \notin q}{\Gamma \vdash p, x <: p, q}\ (\textsc{q-qvar})$$

$$\frac{\Gamma \vdash q_1 <: q_2}{\Gamma \vdash p, q_1 <: p, q_2}\ (\textsc{q-cong}) \qquad \frac{x : T^{q} \in \Gamma \quad \blacklozenge \notin q}{\Gamma \vdash x <: q}\ (\textsc{q-var}) \qquad \frac{\Gamma \vdash p <: q \quad \Gamma \vdash q <: r}{\Gamma \vdash p <: r}\ (\textsc{q-trans})$$

$$\frac{X^{x} <: T^{q} \in \Gamma}{\Gamma \vdash X <: T}\ (\textsc{s-tvar}) \qquad \frac{}{\Gamma \vdash T <: \top}\ (\textsc{s-top}) \qquad \frac{\Gamma \vdash S <: T \quad \Gamma \vdash p <: q}{\Gamma \vdash S^{p} <: T^{q}}\ (\textsc{sq-sub})$$

$$\frac{\Gamma \vdash \alpha_1 <: \alpha_2 \quad \Gamma \vdash \beta_1 <: \beta_2}{\Gamma \vdash \{u : \alpha_1; k : \beta_1\} <: \{u : \alpha_2; k : \beta_2\}}\ (\textsc{e-sub}) \qquad \frac{\Gamma \vdash S^{p} <: T^{q} \quad \Gamma \vdash \varepsilon_1 <: \varepsilon_2}{\Gamma \vdash S^{p}\,\varepsilon_1 <: T^{q}\,\varepsilon_2}\ (\textsc{sqe-sub})$$

Fig. 3. Subtyping rules of $\mathsf{F}^{\blacklozenge}_{\varepsilon<:}$.

For this reason, it is also safe to regard effect qualifiers as non-fresh. Since fresh resources are untracked, and cannot be observed by the subsequent program unless they are returned from a function application. Therefore, as long as we control the returning and killing of fresh resources at function boundaries, we can safely ignore the freshness component in effect qualifiers without losing soundness.

Both application rules enforce a similar constraint when the function captures fresh resources ($\blacklozenge \in p$): the function body must not *both return and kill* the function self reference.

Type applications are similar, except that they do not evaluate an argument term, and thus do not have the corresponding argument effect ε_2.

3.2.7 Subsumption. T-SUB permits upcasting terms to a supertype and with a coarser set of effects, as long as the qualifier and the effects are observable in the current context (that is, $q, \varepsilon_2 \subseteq \varphi, \blacklozenge$).

3.3 Subtyping

3.3.1 Core Type Subtyping. S-BASE states reflexivity for base types: each ground type is a subtype of itself. S-TRANS provides transitivity, allowing multi-step derivations to collapse into a single subtyping step.

3.3.2 Reference Types. S-REF makes reference types *invariant* in their payload: it demands both $S <: T$ and $T <: S$. This rules out unsound variance for writable references. Intuitively, two reference types are related by subtyping only when their payloads are mutually subtypes. Subtyping does not grant additional read/write capability.

3.3.3 Function Types. S-FUN gives the standard contravariance in the parameter and covariance in the result, extended to our effect annotations. The premise $P <: O$ captures contravariant function arguments, and the arguments must be pure. The body check is performed under a context extended with a fresh self name f and parameter $x : P$. Both the qualified type of the result and the latent effect are checked covariantly under this context.

3.3.4 Qualifier Subtyping. Q-SUB introduces qualifier subtyping by inclusion, bounded by the fresh name and the environment's domain. Q-CONG is a congruence principle: extending the right-hand component of two qualifiers preserves subtyping. Q-SELF allows upcasting any function's captured resource to itself, which is useful when reasoning about escaping closures. Q-VAR traces the reachability chain and allows variables to be upcast to their reachable qualifiers. Both Q-SELF and Q-VAR require the qualifier in context to be non-fresh. Q-QVAR lifts bounds on qualified type variables, and Q-TRANS provides transitivity for qualifiers, composing subtyping steps.

3.3.5 Type Variables and Top. S-TVAR reads bounds from the context: if the environment contains $X <: T$, then X is a subtype of T. S-TOP introduces the top type: every type is a subtype of $\top$.

3.3.6 Qualified Types. SQ-SUB combines subtyping of the underlying types with subtyping of their qualifiers: if $S <: T$ and $p <: q$, then $S^p <: T^q$. The rule is pointwise and imposes no further interaction beyond well-formedness of the qualifiers involved.

3.3.7 Effects. E-SUB defines effect subtyping componentwise. With effects being represented as a pair (a use component α and a kill component β), subtyping requires each component to be in a sub-qualifier relation.

3.3.8 Qualified Types with Effects. SQE-SUB is the combined subtyping judgment over type, qualifier, and effects. Each component must advance in its own subtyping relation: the underlying type by type subtyping, the qualifier by qualifier subtyping, and the effects by effect subtyping.

3.4 Dynamics

The dynamic semantics of $\mathsf{F}^{\blacklozenge}_{\varepsilon<:}$ follows the standard call-by-value reduction for the λ-calculus with mutable references. A runtime configuration pairs a store σ with an expression, and reduction steps are taken in the evaluation contexts from Figure 4. Stores map locations to either a value or the distinguished marker $\dagger$ denoting deallocated states, which is essential for soundness with qualified types. The store typing judgments in the figure ensure that every allocated location is described by a well-formed qualified reference type and that any location visible to the observer set φ is inhabited by a value consistent with its typing annotation.

The core reduction rules are those of the traditional CBV calculus. Rule STEP-BETA performs term-level function application by substituting the argument value for the parameter and the closure for its self-reference. Reference allocation STEP-REF extends the store with a fresh location whose content is the allocated value; STEP-DEREF reads the value stored at a reachable location; and STEP-ASSIGN overwrites the content of an existing location while preserving the overall store domain. Because evaluation contexts enforce left-to-right evaluation, these operations align with the sequencing discipline required by the static effect system.

The highlighted rules STEP-FREE and STEP-MOVE capture the destructive effects that distinguish $\mathsf{F}^{\blacklozenge}_{\varepsilon<:}$. Deallocation **free** ℓ replaces the content of the reference cell corresponding to ℓ with $\dagger$. Note that it does that regardless of whether the cell is already deallocated or not so that the deallocation operation is *idempotent* (See Section 2.2.7). Moving **move** ℓ, on the other hand, is not idempotent, and requires a live location, then allocates a fresh location w that stores the payload while marking the original cell as $\dagger$. The term-level result is the fresh location, which matches the typing rule that returns the moved value at a fresh qualifier. The freshness side condition in STEP-MOVE enforces the same dynamic discipline as the static **move** typing rule: a moved reference cannot be accessed again through its original pointer, yet the payload remains available through a newly allocated name.

Together, these rules implement the effectful behaviours tracked in Figure 2. Every use effect corresponds to reading or writing a live location, whereas kill effects materialize exactly in the

Term Typing (with Store Typing) $\boxed{[\Gamma \mid \Sigma]^{\varphi} \vdash t : Q}$

ℓ	$\in$	Loc	Locations
t	$::=$	$\cdots \mid \ell$	Terms
v	$::=$	$\lambda f(x).t \mid c \mid \ell \mid \mathbf{unit} \mid \Lambda f(X^{\chi}).t$	Values
p, q, r	$\in$	$\mathcal{P}_{\text{fin}}(\text{Var} \uplus \text{Loc} \uplus \{\blacklozenge\})$	Qualifiers
φ	$\in$	$\mathcal{P}_{\text{fin}}(\text{Var} \uplus \text{Loc})$	Observations
Σ	$::=$	$\varnothing \mid \Sigma, \ell : T^{q,x}$	Store Typing

Store Bindings, Stores, Store Contexts, Evaluation Contexts

$$v_{\dagger} ::= v \mid \dagger \qquad \sigma ::= \overline{\ell : v_{\dagger}} \qquad S ::= \sigma \,\square\, \sigma$$
$$E ::= \square \mid E\ t \mid v\ E \mid \mathbf{ref}\ E \mid !E \mid E := t \mid v := E \mid \mathbf{free}\ E \mid \mathbf{move}\ E$$

Well-Formed and Well-Typed Stores $\boxed{\Gamma \mid \Sigma \vdash \sigma}$ $\boxed{\Sigma\ \text{ok}}$

$$[\Gamma \mid \Sigma]^{\varphi} \vdash \sigma \ := \ \varphi \subseteq \text{dom}(\sigma) \subseteq \text{dom}(\Sigma) \wedge \forall \ell \in \varphi, [\Gamma \mid \Sigma]^{\varphi} \vdash \sigma(\ell) : \Sigma(\ell)$$
$$\Gamma \mid \Sigma \vdash \sigma \ := \ [\Gamma \mid \Sigma]^{\text{dom}(\Sigma)} \vdash \sigma$$

$$\frac{}{\varnothing\ \text{ok}}\ (\textsc{st-emp}) \qquad \frac{\Sigma\ \text{ok} \quad \text{fv}(T) = \varnothing \quad \text{ftv}(T) = \varnothing \quad q \in \text{dom}(\Sigma) \quad \ell \notin \text{dom}(\Sigma)}{\Sigma, \ell : T^{q}\ \text{ok}}\ (\textsc{st-con})$$

Reduction Rules $\boxed{\sigma \mid t \longrightarrow \sigma \mid t}$

$$\sigma \mid E[\ (\lambda f(x).t)\ v\ ] \longrightarrow \sigma \mid E[\ t[v/x, (\lambda f(x).t)/f]\ ] \quad (\textsc{step-}\beta)$$
$$\sigma \mid E[\ \mathbf{ref}\ v\ ] \longrightarrow \sigma, \ell : v \mid E[\ \ell\ ], \quad \ell \notin \text{dom}(\sigma) \quad (\textsc{step-ref})$$
$$S[\ \ell : v\ ] \mid E[\ !\ell\ ] \longrightarrow S[\ \ell : v\ ] \mid E[\ v\ ] \quad (\textsc{step-deref})$$
$$S[\ \ell : v\ ] \mid E[\ \ell := v'\ ] \longrightarrow S[\ \ell : v'\ ] \mid E[\ \mathbf{unit}\ ] \quad (\textsc{step-assign})$$
$$S[\ \ell : v_{\dagger}\ ] \mid E[\ \mathbf{free}\ \ell\ ] \longrightarrow S[\ \ell : \dagger\ ] \mid E[\ \mathbf{unit}\ ] \quad (\textsc{step-free})$$
$$S[\ \ell : v\ ] \mid E[\ \mathbf{move}\ \ell\ ] \longrightarrow S[\ \ell : \dagger\ ]\ w : v \mid E[\ w\ ], \quad w \notin \text{dom}(\sigma) \quad (\textsc{step-move})$$

Fig. 4. Store Typing, Well-Formed Stores, and Operational Semantics for $\mathsf{F}^{\blacklozenge}_{\varepsilon<:}$. The semantics for **free** and **move** are highlighted in gray boxes.

reductions that insert $\dagger$ into the store. The operational semantics therefore mirrors the effect annotations used in the preservation proof of Section 3.5, allowing the metatheory to relate static budgets to the concrete run-time mutations of the heap.

3.5 Metatheory

Theorem 3.1 (Preservation).

$$\frac{\begin{array}{c}[\Sigma \mid \varnothing]^{\text{dom}(\Sigma)} \vdash t : T^{q}\ \varepsilon \qquad \sigma \mid t \longrightarrow \sigma' \mid t' \\ \Sigma \mid \varnothing\ \text{ok} \qquad [\Sigma \mid \varnothing]^{\text{dom}(\Sigma)} \mid k \vdash \sigma \qquad \Gamma \vdash \{\mathcal{k} : k\} \rhd \varepsilon \end{array}}{\begin{array}{c}\text{dom}(\sigma) \subseteq \text{dom}(\sigma') \quad \exists \Sigma', \varepsilon', \varepsilon''\ p,\ k', \varphi'.\quad \varepsilon'' \subseteq \varepsilon \quad \Sigma' \supseteq \Sigma \quad \varphi \subseteq \varphi' \quad p \subseteq \varphi' \\ \Sigma' \mid \varnothing\ \text{ok} \quad k' \subseteq \mathcal{k}(\varepsilon) \quad [\Sigma' \mid \varnothing]^{\text{dom}(\Sigma')} \mid k, k' \vdash \sigma' \quad \exists p, \mathcal{u}(\varepsilon'), \mathcal{k}(\varepsilon') \subseteq \text{dom}(\Sigma') \setminus \text{dom}(\Sigma). \\ \mathcal{k}(\varepsilon') \cap p \subseteq \varnothing \quad \Gamma \vdash \{\mathcal{k} : k, k'\} \rhd \varepsilon, \varepsilon' \quad [\Sigma' \mid \varnothing]^{\text{dom}(\Sigma')} \vdash t' : T^{q[p/\blacklozenge]}\ \varepsilon'' \rhd \varepsilon' \end{array}}$$

Preservation (intuition). The step $\sigma \mid t \longrightarrow \sigma' \mid t'$ preserves typing by growing (never shrinking) the heap and refining the static world: there is $\Sigma' \supseteq \Sigma$ with $\Sigma' \mid \varnothing\ \text{ok}$ and a concrete fresh path p that instantiates the abstract freshness in the type, yielding $[\Sigma' \mid \varnothing]^{\text{dom}(\Sigma')} \vdash t' : T^{q[p/\blacklozenge]}\ \varepsilon'' \rhd \varepsilon'$. Effects split into the already-accounted residual $\varepsilon'' \subseteq \varepsilon$ and the step-local payload ε', which may use

Qualifier Transitive Lookup $\boxed{\Gamma \vdash q^n}$ $\boxed{\Gamma \vdash q*}$

$\Gamma \vdash q^0 := q$ Qualifier Transitive Lookup Base Case

$\Gamma \vdash q^{n+1} := ((\bigcup_{x \in q} \{ y \mid x \rightsquigarrow y \}) \cup q)^n$ Qualifier Transitive Lookup Inductive Case

$\Gamma \vdash q* := q^{|\Gamma|}$ Qualifier Full Transitive Lookup

Qualifier Notations $\boxed{\Gamma \vdash \mathrm{dsat}\ q}$ $\boxed{\Gamma \vdash p \pitchfork q}$

$\Gamma \vdash \mathrm{dsat}\ q := q* = q$ Qualifier Saturation $\quad \Gamma \vdash p \pitchfork q := \blacklozenge, (p* \cap q*)$ Qualifier Overlap

Effect Composition $\boxed{\Gamma \vdash \varepsilon_1 \rhd \varepsilon_2}$

$$\frac{\Gamma \vdash \not\hbar(\varepsilon_1) \pitchfork u(\varepsilon_2) = \varnothing}{\Gamma \vdash \varepsilon_1 \rhd \varepsilon_2 = \{u : (u(\varepsilon_1) \cup u(\varepsilon_2)); \not\hbar : (\not\hbar(\varepsilon_1) \cup \not\hbar(\varepsilon_2))\}} \text{(EFF-SEQ)}$$

Fig. 5. Operations on effects. Sequential effect composition is the component-wise set union and partial. It is only defined if no variable/location is used after being killed.

or kill only newly allocated locations in $\mathrm{dom}(\Sigma') \setminus \mathrm{dom}(\Sigma)$ and never kill p itself ($\not\hbar(\varepsilon') \cap p \subseteq \varnothing$). The store obligations are threaded by adding a bounded compensation k' with $k' \subseteq \epsilon.k$ (i.e., $k' \subseteq \not\hbar(\varepsilon)$), so that $[\Sigma' \mid \varnothing]^{\mathrm{dom}(\Sigma')} \mid k, k' \vdash \sigma'$ and $\{\not\hbar : k, k'\} \rhd \varepsilon, \varepsilon'$ hold. Intuitively, any dynamically killed resource at this step must already be permitted by the static kill component; we shrink the static budget by removing the effects that actually occurred (moving them into ε'), and the remaining static effects may be realized in later steps—or not at all—since effects are an overapproximation. Finally, the typing rule for fresh application forbids simultaneously returning and killing the freshly created location in the same step, which would otherwise lose the witness for reachability and break subject reduction.

Proof sketch. By induction on the typing derivation of t. All structural cases follow by the induction hypothesis and the congruence rules, threading world extension and effect splitting. The interesting case is application. For $t = t_1\ t_2$, analyze whether the argument (or self) is fresh and whether the function returns or kills that argument: (i) if the argument is not fresh and the function returns it, no kill occurs and we instantiate the result qualifier as $q[p/\blacklozenge]$ with p absent, taking ε' to contain only the immediate uses; (ii) if the argument is fresh and the function returns it, we produce a new $p \subseteq \varphi'$ and instantiate $q[p/\blacklozenge]$, with $\not\hbar(\varepsilon') \cap p \subseteq \varnothing$ by rule T-APP-FRESH; (iii) if the function kills the (non-fresh) argument, we set $k' \subseteq \epsilon.k$ to cover exactly the aliases dynamically killed this step and place those kills in ε'; (iv) the combination "fresh and killed" is ruled out statically by t-app-fresh, ensuring no step both introduces and discards the new path. In each subcase, choose Σ' to account for new allocations, pick ε'' as the residual static budget ($\varepsilon = \varepsilon'' \rhd \varepsilon'$), and conclude the reduct typing and store well-formedness as required.

3.6 Semantic Type Soundness

In addition to proving syntactic type soundness, we define unary logical relations for a variant of the $\mathsf{F}^{\blacklozenge}_{\varepsilon<:}$-calculus with kill effect, and prove semantic type soundness (the fundamental property). For simplicity, the variant excludes subtyping and cyclic references, following prior modeling of those features in other settings [Bao et al. 2025]. Interested readers can find our Rocq mechanization online.

4 Limitations and Future Work

Our current operational model implements both **move** and swap by allocating a fresh destination location and deallocating the source that was moved or swapped away. While this faithfully captures logical ownership transfer, it does not reflect a true in-place swap at the physical memory level. Supporting such operations would require breaking the telescoping property of the store typing and introducing a virtual memory layer capable of observing alias reshuffling directly.

To see the semantic tension concretely, consider swapping through a nested reference:

```
val x  = new Ref(1)          // : Ref[Int]^x
val y  = new Ref(2)          // : Ref[Int]^y
val nc = new Ref(move y)     // : Ref[Ref[Int]^∅]^nc ; {𝓊 : y; ℛ : y}
val z  = swap(nc, x)         // : Ref[Int]^z ; {ℛ : x} ▷ {𝓊 : nc}
```

Allocating the outer cell nc moves ownership of y, inducing a kill effect on the entire reachability closure of y. Swapping through nc then produces z while simultaneously killing x and writing to nc. Our present operational model simulates these kills by deallocating and reallocating references rather than performing an in-place update, leaving true physical swaps to future work.

The swap idiom is therefore limited to storing references (possibly nested) in the inner position. This restriction ensures that the inner element is unique at the time of the swap: once it is moved out, there are no remaining aliases that could observe or duplicate the "fresh" resource. If the inner value were instead a higher-order value such as a function, the function could close over multiple locations. Supporting the swap would then require moving every captured location transitively through the closure, a capability our current effect system does not provide. In particular, we lack a way to disable access paths through the store that are only indirectly reachable via such higher-order values.

One possible path forward is to introduce a *virtual memory* layer. Rather than allocating a new physical reference on every swap, this layer would maintain the telescoping invariant on virtual names and perform renaming on the logical store. Each virtual name would map one-to-one onto a live location in the physical heap, preserving heap topology while allowing the logical swap to repoint aliases without data movement. Such a design would more faithfully model an in-place swap, avoid spurious allocation, and potentially generalize to inner elements with richer structure.

In summary, the current system guarantees safety by allocating fresh locations for each swap and by restricting the inner value to references, but consequently cannot handle arbitrary types nor suppress aliasing effects transitively through the store. Eliminating the fresh-allocation requirement and supporting richer inner values remain important directions for future work.

5 Related Work

Reachability Types. Bao et al. [2021] introduced RT that support only first-order mutable references. Wei et al. [2024] extends their work to support polymorphism and higher-order stores.

Deng et al. [2025] introduces cyclic reference types, enabling recursive constructs without built-in mechanisms. They extend non-cyclic references to generalize the semantics introduced in Wei et al. [2024], providing additional flexibility for cyclic structures. They also refine the reference introduction rule to increase precision and allow flexible graph structures with nested references.

He et al. [2025] unifies region/ownership-style lifetime control with reachability types by introducing *arenas* that permit arbitrary in-arena sharing while guaranteeing lexically scoped lifetimes, while also allowing individually managed resources within the enclosing arena, with guaranteed deallocation at the end of the arena's lifetime. As the first reachability-based formalism for lifetime control, it avoids flow-sensitive reasoning.

In contrast, our system integrates reachability reasoning with an explicit, flow-sensitive effect discipline that supports destructive updates. Rather than restricting deallocation to arena boundaries, we refine an effect system on top of RT with *use*, and *kill* effects that tracks and controls the operations performed on resources. This enables precise tracking of aliasing across higher-order functions, proves use-after-free freedom for cyclic structures, and permits ownership transfer without imposing an arena discipline.

Additionally, Bao et al. [2025] formalize RT using logical relations to prove key properties, including termination even in the presence of higher-order store, which is a key premise for our work to build on. Graph IR [Bračevac et al. 2023] leverages RT to optimize impure higher-order programs by tracking fine-grained dependencies with an effect system. Jia et al. [2024] address key challenges in self-references by proposing an enhanced notion of subtyping and developing a sound and decidable bidirectional typing algorithm for RT.

Separation and alias control. Separation Logic enables local heap reasoning via separating conjunction [Reynolds 2002]. RT adopts this idea in its application rule by requiring disjointness between function and argument qualifiers [Bao et al. 2021; Wei et al. 2024]. For concurrency, CSC extends CC to enforce static separation and race freedom [Boruch-Gruszecki et al. 2023; Xu et al. 2023, 2024], and separation logic has been combined with effect handlers for cooperative concurrency and mutable state [De Vilhena and Pottier 2021]. Verona's Reggio uses a forest of regions with a single window of mutability per thread and per-region strategies, with asynchronous cowns for isolated sharing [Arvidsson et al. 2023]. In contrast, we enforce separation and effect safety by extending RT with destructive effects and move semantics without region windows or proof obligations.

Regions, lifetimes, and deallocation. Classic region calculi provide static memory management with explicit effects and safe deallocation [Lippmeier 2013]; more recently, implicit, non-lexical, splittable regions with sized allocations are enforced by an effect system rather than substructural types [Hughes et al. 2025]. Region inference historically pursued early free and late allocation via whole-program constraints [Aiken et al. 1995; Tofte and Talpin 1994, 1997]. RT instead operates at alias level: kill effects deallocate exactly the reachable alias set of a reference and, together with move, prevent use-after-free without imposing a region discipline, while still recovering region-style guarantees by layering effects over reachability [Bao et al. 2021].

Ownership, linearity, and uniqueness. Separation and safety for concurrent programs can be obtained by combining linear typing with regions [Milano et al. 2022]. Capability systems support at-most-once consumption with flexible borrowing [Haller and Odersky 2010]. Linear Haskell brings linearity to a higher-order, pure setting [Bernardy et al. 2018]. A comparative account clarifies trade-offs between linearity and uniqueness [Marshall et al. 2022]. Fully in-place execution for a class of pure programs is captured via a linear calculus with embeddings using uniqueness or precise reference counting [Lorenzen et al. 2023]. In Rust, local ownership permits multiple mutable aliases within thread-local scopes to support cycles [Noble et al. 2023]. RT differs by not imposing global substructural disciplines or lexical ownership: uniqueness and transfer arise as modes of use governed by flow-sensitive kill and move on reachability qualifiers within shared, higher-order code.

Capabilities, capture, and coeffects. Scoped capabilities type captured variables by extending $F_{<:}$ with capture sets, subcapturing, and boxing (Scala 3 prototype) [Odersky et al. 2022]. Capturing Types track captured capabilities in Scala, drawing on contextual modal type theory [Boruch-Gruszecki et al. 2023; Nanevski et al. 2008]. Non-structural coeffects track reduction-induced sharing and validate modifiers and capsule invariants [Bianchini et al. 2022]; heterogeneous coeffects and

coeffect classes enable safe composition [Bianchini et al. 2023]. RT instead exposes aliasing through reachability qualifiers and controls consumption via destructive use, kill, and move, focusing on when resources may be used or deallocated rather than which capabilities are captured or which coeffects are present.

DOT, path dependence, and qualified types. DOT formalizes Scala's path-dependent types and recursive self types with restrictions on dependent application [Rompf and Amin 2016]; extensions add reference mutability and purity-oriented function types [Dort and Lhoták 2020; Dort et al. 2024]. Qualified types underpin const inference, Java reference immutability, and qualified polymorphism [Foster et al. 2006; Huang et al. 2012; Lee and Lhoták 2023]. RT repurposes qualifiers to encode heap reachability and separation, and augments them with destructive effects to enforce move and explicit deallocation—capabilities not addressed by these DOT and qualification frameworks.

6 Conclusion

We have shown how reachability types can be extended with a flow-sensitive effect system that captures explicit memory management idioms without sacrificing higher-order expressiveness. By distinguishing *use* effects for reads and writes from *kill* effects for destructive actions, the system provides precise summaries of how programs interact with resources, enabling ownership transfer, contextual freshness, and memory reclamation to coexist in a single calculus. The operational semantics and mechanized soundness proof establish that well-typed programs avoid use-after-free faults while supporting idioms such as safe deallocation, move semantics, and swap abstractions. Our case studies suggest that these guarantees scale to realistic examples while keeping annotations manageable.

Looking forward, we plan to explore richer concurrency scenarios, integrate the effect system with region-aware runtime implementations, and investigate more expressive alias reshuffling primitives that preserve the telescoping structure of the store typing. These directions would further bridge the gap between theoretical foundations and the practical demands of safe manual memory management in functional languages.

ACKNOWLEDGEMENTS

This work was supported in part by NSF awards 2348334, Augusta faculty startup package, as well as gifts from Meta, Google, Microsoft, and VMware.

Data Availability Statement

Rocq mechanizations can be found at https://github.com/tiarkrompf/reachability.

References

Alexander Aiken, Manuel Fähndrich, and Raph Levien. 1995. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. In *PLDI*. ACM, 174–185.

Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management. *Proc. ACM Program. Lang.* 7, OOPSLA2 (Oct. 2023), 270:1363–270:1393. doi:10.1145/3622846

Yuyan Bao, Songlin Jia, Guannan Wei, Oliver Bračevac, and Tiark Rompf. 2025. Modeling Reachability Types with Logical Relations: Semantic Type Soundness, Termination, and Equational Theory. *Proc. ACM Program. Lang.* 9, OOPSLA (2025). doi:10.1145/3763116

Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability Types: Tracking Aliasing and Separation in Higher-Order Functional Programs. *Proc. ACM Program. Lang.* 5, OOPSLA (Oct. 2021), 139:1–139:32. doi:10.1145/3485516

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL (2018), 5:1–5:29. doi:10.1145/3158093

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2023. A Java-like Calculus with Heterogeneous Coeffects. *Theoretical Computer Science* 971 (Sept. 2023), 114063. doi:10.1016/j.tcs.2023.114063

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, and Marco Servetto. 2022. Coeffects for Sharing and Mutation. *Proc. ACM Program. Lang.* 6, OOPSLA2 (Oct. 2022), 156:870–156:898. doi:10.1145/3563319

Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.* 45, 4 (Nov. 2023). doi:10.1145/3618003

Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM Program. Lang.* 7, OOPSLA2 (Oct. 2023), 236:400–236:430. doi:10.1145/3622813

David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. ACM, 48–64.

Paulo Emílio De Vilhena and François Pottier. 2021. A Separation Logic for Effect Handlers. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 1–28. doi:10.1145/3434314

Haotian Deng, Siyuan He, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2025. Complete the Cycle: Reachability Types with Expressive Cyclic References (Extended Version). arXiv:2503.07328 [cs] doi:10.48550/arXiv.2503.07328

Vlastimil Dort and Ondřej Lhoták. 2020. Reference Mutability for DOT. In *LIPIcs, Volume 166, ECOOP 2020*, Vol. 166. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 18:1–18:28. doi:10.4230/LIPICS.ECOOP.2020.18

Vlastimil Dort, Yufeng Li, Ondřej Lhoták, and Pavel Parízek. 2024. Pure Methods for roDOT. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 13:1–13:29. doi:10.4230/LIPIcs.ECOOP.2024.13

Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. 2006. Flow-Insensitive Type Qualifiers. *ACM Transactions on Programming Languages and Systems* 28, 6 (Nov. 2006), 1035–1087. doi:10.1145/1186632.1186635

Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer, Berlin, Heidelberg, 354–378. doi:10.1007/978-3-642-14107-2_17

Siyuan He, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2025. When Lifetimes Liberate: A Type System for Arenas with Higher-Order Reachability Tracking. arXiv:2509.04253 [cs] doi:10.48550/arXiv.2509.04253

Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. 2012. Reim & ReImInfer: Checking and Inference of Reference Immutability and Method Purity. *SIGPLAN Not.* 47, 10 (Oct. 2012), 879–896. doi:10.1145/2398857.2384680

Jack Hughes, Michael Vollmer, and Mark Batty. 2025. Spegion: Implicit and Non-Lexical Regions with Sized Allocations. arXiv:2506.02182 [cs] doi:10.48550/arXiv.2506.02182

Songlin Jia, Guannan Wei, Siyuan He, Yuyan Bao, and Tiark Rompf. 2024. Escape with Your Self: A Solution to the Avoidance Problem with Decidable Bidirectional Typing for Reachability Types. arXiv:2404.08217 [cs] doi:10.48550/arXiv.2404.08217

Edward Lee and Ondřej Lhoták. 2023. Simple Reference Immutability for System F $_{<:}$. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (Oct. 2023), 857–881. doi:10.1145/3622828

Ben Lippmeier. 2013. Mechanized Soundness for a Type and Effect System with Region Deallocation. (2013).

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. *Proceedings of the ACM on Programming Languages* 7, ICFP (Aug. 2023), 275–304. doi:10.1145/3607840

Danielle Marshall, Michael Vollmer, and Dominic Orchard. 2022. Linearity and Uniqueness: An Entente Cordiale. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Vol. 13240. Springer International Publishing, Cham, 346–375. doi:10.1007/978-3-030-99336-8_13

Mae Milano, Joshua Turcotti, and Andrew C. Myers. 2022. A Flexible Type System for Fearless Concurrency. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 458–473. doi:10.1145/3519939.3523443

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *ACM Transactions on Computational Logic* 9, 3 (2008), 23:1–23:49. doi:10.1145/1352582.1352591

James Noble, Julian Mackay, and Tobias Wrigstad. 2023. Rusty Links in Local Chains✱. In *Proceedings of the 24th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '22)*. Association for Computing Machinery, New York, NY, USA, 1–3. doi:10.1145/3611096.3611097

Martin Odersky, Aleksander Boruch-Gruszecki, Edward Lee, Jonathan Brachthäuser, and Ondřej Lhoták. 2022. Scoped Capabilities for Polymorphic Effects. arXiv:2207.03402 [cs] doi:10.48550/arXiv.2207.03402

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. doi:10.1109/LICS.2002.1029817

Tiark Rompf and Nada Amin. 2016. Type Soundness for Dependent Object Types (DOT). *SIGPLAN Not.* 51, 10 (Oct. 2016), 624–641. doi:10.1145/3022671.2984008

Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the Typed Call-by-Value Lambda-Calculus Using a Stack of Regions. In *POPL*. ACM Press, 188–201.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (Feb. 1997), 109–176. doi:10.1006/inco.1996.2613

Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 393–424. doi:10.1145/3632856

Yichen Xu, Aleksander Boruch-Gruszecki, and Martin Odersky. 2023. Degrees of Separation: A Flexible Type System for Data Race Prevention. IWACO (2023). doi:10.48550/ARXIV.2308.07474

Yichen Xu, Aleksander Boruch-Gruszecki, and Martin Odersky. 2024. Degrees of Separation: A Flexible Type System for Safe Concurrency. *Proc. ACM Program. Lang.* 8, OOPSLA1 (April 2024), 136:1181–136:1207. doi:10.1145/3649853