When Lifetimes Liberate: A Type System for Arenas with Higher-Order Reachability Tracking

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

Statically enforcing safe resource management is challenging due to tensions between flexible lifetime disciplines and expressive sharing patterns. Region-based systems offer lexically scoped regions under a stack discipline, wherein resources are managed in bulk. In many such systems, however, resources are second-class and can neither escape their scope nor be freely returned from functions. Ownership and linear type systems, such as Rust, offer first-class, non-lexical lifetimes with robust static guarantees, but rely on invariants that limit higher-order patterns and expressive sharing.

In this work, we propose a type system that uniformly treats all heap-allocated resources under diverse lifetime, granularity, and sharing settings. Our system provides programmers with three allocation modes: (1) fresh allocation for first-class, non-lexical resources; (2) fresh allocation for second-class resources with lexically bounded lifetimes; and (3) coallocation that groups resources by shadow arenas for bulk tracking and deallocation. Regardless of mode, resources are represented uniformly at the type level, supporting generic abstraction and preserving the higher-order parametric nature of the language.

Obtaining static safety in higher-order languages with flexible sharing is nontrivial. To address this, our solution builds on reachability types, and our extension adds the capability to track both individual and grouped resources, enables the expression of cyclic store structures, and allows the selective enforcing of stack lifetime discipline. These mechanisms are formalized in the $A^{\blacklozenge}_{<:}$ and $\{A\}^{\blacklozenge}_{<:}$ type systems, which are proven type safe and memory safe in Rocq.

1 Introduction

Higher-order functional languages, starting with LISP [McCarthy 1959], typically rely on garbage collection or reference counting for automatic resource management. Although safe and convenient, these approaches offer little control over the timing or granularity of deallocation. Such a trade-off is often acceptable for memory resources, but is a key limitation for system assets like files, sockets, or large heaps. Languages mitigate this trade-off with explicit APIs like `FileReader.close` in Java or scoped constructs like `with` blocks in Python. Still, these patterns do not ensure safety *statically*: deallocated resources may be referenced directly or leaked via global storage, leading to runtime exceptions at best and data corruption at worst in *use-after-free* (UAF) situations. An ideal solution should combine the convenience of automatic management with explicit control for critical resources, enforcing static safety like no UAF without restricting common higher-order expressiveness.

Stack and Heap Allocation. To recognize the challenge, recall the classic contrast between stack and heap allocation. Stack values have strictly lexical lifetimes and are reclaimed automatically when their scope ends, guaranteeing safety and efficiency, but making them inherently *second-class* where they cannot escape their defining scope, be returned, or be stored in a heap structure. In contrast, heap allocation yields *first-class* values that can flow freely through higher-order functions,

Authors' Contact Information: 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.

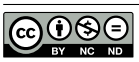


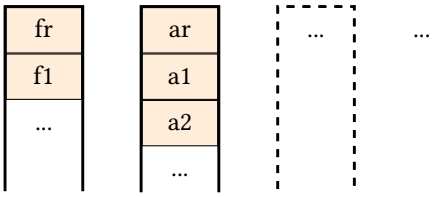
Table 1. Comparison of resource tracking/management solutions. Each solution emphasizes a distinct strategy, while in this work, we try to unify their strengths on control, expressiveness, and flexibility.

Features	Cyclone[a] [LIFO]	Cyclone[b] [Dynamic]	Safe Rust[c] [Ownership]	Reggio[d] [Region+OT]	$F^{\blacklozenge}_{<:}$[e] [Reachability]	This Work [Region+RT]
Stack Lifetime Discipline	✓	✓	✓	✓		✓
Mutable, Cyclic Sharing	✓	✓		✓		✓
First-Class Escaping		✓	✓	✓	✓	✓
Higher-Order Functions	✓		✓		✓	✓

[a]Grossman et al. [2002] [b]Hicks et al. [2004] [c]Klabnik and Nichols [2019] [d]Arvidsson et al. [2023] [e]Wei et al. [2024]

```
{
  val fr = new Ref(42)          //: Ref[Int]^fr
  val ar = new Ref(42) scoped   //: Ref[Int]^ar
  val f1 = new Ref(42) at fr    //: Ref[Int]^f1,fr
  val a1 = new Ref(42) at ar    //: Ref[Int]^a1,ar
  val a2 = new Ref(42) at a1    //: Ref[Int]^a2,a1
}  // guaranteed to deallocate {ar,a1,a2}
```

(a) An allocation can take one of three forms: primitive (`fr`), scoped (`ar`), and coallocation (`f1,a1,a2`). Scoped arenas (*e.g.*, `ar`) are deallocated when leaving the scope.



(b) The two-dimensional layout produced by (a). Two arenas are created; they can be referred to in **`at`** clauses through existing cells, *e.g.*, `fr,ar,a1`.

Fig. 1. Illustration of the three allocation forms and the corresponding two-dimensional store layout.

either by being captured in closures or shared across structures. However, such flexibility comes at the cost of timely deallocation by programmer control, while garbage collection or reference counting eventually reclaim objects, giving up the static safety of stack discipline.

Works such as Osvald et al. [2016], which reintroduce second-class values in higher-order programs to provide memory safety and scoped lifetime control, have also been shown to be effective at reducing GC pressure [Xhebraj et al. 2022]. More recent efforts have revived interest in the area, for example, stoic functions [Liu et al. 2020] and a lambda calculus with second-class functions and references [Thiemann 2025]. We aim to combine the expressiveness of first-class heap objects with the static safety of stack allocation.

Regions with Scoped Lifetimes. Region-based systems [Grossman et al. 2002; Tofte et al. 2001] can be viewed as an attempt to re-introduce these stack-like benefits within heap allocation. Programmers allocate groups of memory resources together into regions (arenas), and deallocate an entire region in bulk upon exiting a designated lexical scope, similar to `with` blocks in Python. Values can be shared freely within a region, but the region itself is lexically scoped.

Arenas: When Lifetime Liberate. Lexically scoped regions are not the only model for safe resource management, as individual, non-lexical lifetimes are also beneficial [Fluet and Wang 2004], particularly for patterns like event loops and garbage collectors. Hicks et al. [2004] further recharacterized regions as *LIFO arenas*, highlighting that *arenas*, dynamically growable collections for bulk operation, can integrate diverse lifetime control strategies to additionally achieve *flexibility in lifetimes*. Building on this insight, prior works [Arvidsson et al. 2023; Fluet et al. 2006] formally explored arenas with linear types [Wadler 1990] and ownership types [Clarke et al. 1998; Noble et al. 1998]. These systems employ flow-sensitive reasoning, which necessitates a first-order setting where the global invariants effectively make all resources behave as second-class, restricting higher-order abstraction. To maintain global invariants, users must explicitly *open/enter* an arena before accessing its content, an operation similar to *mutable borrows* in Rust [Klabnik and Nichols 2019] but at a higher level, which ensures safe deallocation without enforcing LIFO lifetimes.

As summarized in Table 1, prior work spans diverse strategies for resource tracking and management. This work aims to unify their strengths: the safety of stack discipline, the expressiveness of higher-order languages, and the treatment of resources as first-class entities.

1.1 Flexible Resource Management in One Coherent System: Bulk and Individual Granularity, Lexical and Unbounded Lifetimes

Our system treats heap-allocated mutable references as the canonical form of resources. These references are first-class values that may be passed, stored, or returned without restriction. Programmers retain control over resource lifetimes and granularity through three allocation forms:

(1) allocation of a first-class, non-lexical reference;
(2) scoped allocation of a lexically bounded reference following stack discipline; and
(3) coallocation, which groups resources collectively into *shadow arenas*.

All three forms (Table 2) coexist under a single uniform reference type `Ref[T]`, so the language does not distinguish resources following stack discipline from first-class resources at the type level. Built for a higher-order functional setting, our system supports shared mutable state and cyclic store structures without requiring flow-sensitive reasoning.

We illustrate our resource allocation forms in Figure 1a. The **`new`** allocations for `fr` and `ar` introduce two distinct references, and implicitly two distinct shadow arenas to host them. Subsequent coallocations, `f1`, `a1`, and `a2`, coallocate additional resources in the same arena of placement clauses prefixed by **`at`**, avoiding the need for explicit handlers. Scoped allocations, annotated with **`scoped`**, tie a reference and its shadow arena to the lexical scope, enabling automatic bulk deallocation of the entire shadow arena at scope exit. Within their scopes, scoped and non-scoped resources behave uniformly, preserving the first-class abstraction of references while ensuring predictable reclamation where selectively desired.

1.2 Type Safety and Safety of Deallocation in a Higher-Order Setting: Mutable Sharing without Flow-Sensitive Reasoning

In languages with first-class references, higher-order functions, and mutable sharing, statically preventing errors like UAF is challenging. Our system enforces static safety through two complementary mechanisms, corresponding to non-lexical and lexical lifetime control.

First-class safety via reachability tracking. We build on *reachability types* [Bao et al. 2021; Deng et al. 2025a; Wei et al. 2024] to track first-class resources that persist non-lexically. Each type carries a qualifier signifying its reachable resources, marking sharing and separation. We propose $\mathsf{A}^{\blacklozenge}_{<:}$ that extends prior reachability system $\mathsf{F}^{\blacklozenge}_{<:}$ [Wei et al. 2024] with shadow arenas, reference coallocation (**`new at`** in Figure 1a), and a two-dimensional store model (visualized in Figure 1b). $\mathsf{A}^{\blacklozenge}_{<:}$ enables collective tracking and reasoning of resources within an arena. By grouping resources within an arena under a coarse-grained reachability relation, $\mathsf{A}^{\blacklozenge}_{<:}$ achieves greater expressiveness, supporting more complicated store topologies (see Section 2.3.3), while preserving the static safety guarantees as earlier fine-grained approaches.

Selective stack discipline for scoped resources. Programmers may also declare an arena as scoped (**`new scoped`** in Figure 1a), causing it to follow a stack discipline to ensure automatic deallocation at scope exit. Building atop $\mathsf{A}^{\blacklozenge}_{<:}$, $\{\mathsf{A}\}^{\blacklozenge}_{<:}$ introduces such scoped allocation and statically guarantees that there is no UAF error at runtime, while maintaining flow-insensitive reasoning. The scoping control is orthogonal to arenas and generalizable to non-memory resources, yet does not impose any type distinction between first-class and scoped resources.

1.3 Contributions

We propose a unified and flexible system for resource management in higher-order languages with mutable sharing, which combines the strengths of different flavors of resource management. Specifically, our system is novel in the combination of:

(1) Unified treatment of resources: All memory resources are **first-class** values with a **single reference type** that abstracts over stack-like and heap allocation. This allows client code to be generic over the particular storage model of references.
(2) Flexibility in control: Users can manage memory resources lexically or non-lexically, individually or collectively, without introducing type distinctions.
(3) Static safety guarantees: Reachability types track the flow of memory resources and guarantee their safe use. Users can impose selective **stack discipline** to guarantee predictable deallocation and no use-after-free errors through **flow-insensitive** reasoning.
(4) Expressive higher-order features: Our system supports higher-order functions with mutable sharing and cyclic store structures, surpassing the expressiveness of prior reachability systems.

To the best of our knowledge, our system is the first to implement the intersection of these properties in a unified system, without separating first-class and second-class memory resources. It is also the first work to provide a formal deallocation reasoning in this setting, supporting cyclic store structures while preserving static safety guarantees among similar systems.

In this work, we propose a new solution for resource management in higher-order languages with mutable sharing, by integrating *shadow* arenas and scoped lifetime reasoning atop reachability types. Our system offers added flexibility in store topologies compared with previous reachability types, introducing additional scoped or coallocated introduction forms of store objects without type distinction between forms. Specifically,

- After briefly reviewing reachability types, we informally discuss the telescoping structures and the store topology in our system with added flexibility (Section 2).
- We informally overview the $\mathsf{A}^{\blacklozenge}_{<:}$-calculus and the $\{\mathsf{A}\}^{\blacklozenge}_{<:}$-calculus, discussing shadow arenas, coarse-grained reachability tracking, and relaxation of telescoping structures (Section 3).
- We present the formal theory and metatheory of the $\mathsf{A}^{\blacklozenge}_{<:}$-calculus, a reachability type system with non-lexical arenas, coarse-grained reachability tracking, and a two-dimensional store semantics (Section 4).
- We propose the $\{\mathsf{A}\}^{\blacklozenge}_{<:}$-calculus atop $\mathsf{A}^{\blacklozenge}_{<:}$-calculus with scoped arena allocation and flow-insensitive deallocation reasoning, and present its formal theory and metatheory (Section 5).
- We present three case studies, on general fixed point combinators (Section 6.1), callback registration via non-lexical arenas (Section 6.2), and cyclic store structures (Section 6.3).

In Section 7 we reflect on limitations, design choices, and directions for future and parallel work. Section 8 surveys related work, and Section 9 wraps up the paper. All formal results have been mechanized in Rocq, available online at https://github.com/tiarkrompf/reachability .

2 Store Topology and Safety Through Reachability

Before introducing the technical details of our system, we briefly introduce reachability types and discuss how we achieve safety guarantees through reachability tracking. Then we discuss the store topology and store invariants in our system, including a comparison to Rust and previous reachability type systems.

2.1 A Review of Reachability Types

Reachability types [Bao et al. 2021; Deng et al. 2025a; Wei et al. 2024] aim to provide static safety guarantees in higher-order functional programs by tracking aliasing and its absence: separation.

2.1.1 Tracking Resources. The key idea of reachability types is to track which resources a value may reach. A type[1] is annotated with a reachability *qualifier*[2] recording the static *variables* the term may reach. We mark a reference as *fresh* with a marker ♦ in its qualifier if the reference has not yet been bound to a variable. A reference allocation `new Ref(7)` bound to variable `r` is typed as,

```
val r = new Ref(7)          // new Ref(7) : Ref[Int]♦
r                           // r : Ref[Int]^r
```

At the runtime, the `new Ref` expression will evaluate to a fresh *location* in the store, and qualifiers additionally track *locations* in dynamic typing.

```
// new Ref(7) evaluates to some ℓ in the runtime
ℓ                           // ℓ : Ref[Int]^ℓ where σ(ℓ) = 7 in store σ
```

Since the runtime store location of a reference is unknown in the static program, reachability types conservatively approximate runtime locations in static typing. Variables and locations are not necessarily in one-to-one correspondence. For instance, in a conditional statement that reaches references in both branches,

```
val rx = new Ref(42)        // : Ref[Int]^rx
val ry = new Ref(7)         // : Ref[Int]^ry
if (cond) rx else ry        // : Ref[Int]^{rx, ry}
```

The static qualifier of the conditional must conservatively include both `rx, ry`, even though exactly one location will be produced at runtime. In general, a variable in a reachability qualifier represents an *over-approximation* of the set of locations that are actually reached.

2.1.2 Aliases and Closures. Multiple variables can bind to the same value from aliasing.

```
val r = new Ref(7)          // : Ref[Int]^r ⊣ [ r : Ref[Int]♦ ]
val s = r                   // : Ref[Int]^s <: Ref[Int]^r ⊣ [ r : Ref[Int]♦, s : Ref[Int]^r ]
```

Reachability types initially assign each variable a minimal qualifier containing only itself. Through subtyping (Section 4.2), the qualifier of variable `s` can upcast to its recorded qualifier in the typing context, which is `r`, the variable that `s` aliases.

Higher-order functions introduce closures that may capture resources. A function qualifier specifies the resources required for the computation.

```
def f1() = !r                 // : (() => Int)^{f1} ⊣ [ f1 : (() => Int)^r ]
val f2 = {
  val x = new Ref(6)          // : Ref[Int]^x
  () => x                     // : (() => Ref[Int]^x)^x
} // : (() => Ref[Int]^{f2})^{f2}
```

The function qualifier of `f1` in the typing context records the free variables captured by `f1`. Closure `f2` returns a nameless function capturing a locally allocated reference `x`, which loses its name at scope exit and degrades to the self-reference of the closest closure `f2`. Reachability types preserve tracking of resources *escaping* their lexical scopes via function self-references.

2.1.3 Separation and Controlled Sharing. With reachability tracked, functions can require the separation of resources between the argument and the function by the domain qualifier. A freshness marker ♦ in the function domain qualifier requires that the argument is contextually fresh relative to the function, which ensures separation between computations. Controlled sharing can be specified

[1]For bindings in snippets, `val x = t //: T ⊣ Γ`, we illustrate $\Gamma \vdash$ `x : T` unless noted otherwise. Γ might be omitted.
[2]We omit qualifiers for untracked values, including integers and booleans.

through a function domain qualifier ♦q, permitting at most the overlap q observable from both the function and the argument.

```
val u = new Ref(42)                  // : Ref[Int]^u
val v = new Ref(41)                  // : Ref[Int]^v
def f(x : Ref[Int]^♦u) = !u + !v     // : (Ref[Int]^♦u => Int)^u,v
f(u)                                 // OK, {u} ∩ {u,v} ⊆ {♦u}
f(v)                                 // Error: {v} ∩ {u,v}
```

The domain qualifier ♦u of function f permits the argument to share only u with the function, but not v, achieving the sense of controlled sharing.

2.2 Telescoping Structures

As a common issue in dependent type systems, *telescoping structures* arise in reachability types where earlier allocated store objects cannot reach later ones. Such asymmetrical store structures prevent previous reachability type systems from representing cyclic store structures, including mutual recursion.

Telescoping structures are reflected in the static typing, as the order of declared variables approximates the allocation timeline. Consider extending *Landin's Knots* to two mutually recursive references capturing each other,

```
val c1 = ...                     // : Ref[(Int => Int)^q1]^c1
val c2 = ...                     // : Ref[(Int => Int)^q2]^c2
def f1(x : Int) = (!c1)(x)       // : (Int => Int)^c1, captures c1
def f2(x : Int) = (!c2)(x)       // : (Int => Int)^c2, captures c2
c2 := f1                         // OK, q2 can observe c1
c1 := f2                         // Error: q1 cannot observe c2
```

The first assignment `c2 := f1` is type-checked because q2 can include c1, which is in the context at the declaration of c2. The second assignment `c1 := f2` triggers a type error because q1 cannot include c2, as q1 is determined before c2 is declared.

Telescoping structures forbid all store cycles in previous reachability types, preventing the encoding of complex data structures such as cyclic linked lists. In our work, the issue is relaxed through *coarse-grained reachability tracking* that reasons about separation and sharing at the arena level. This permits cyclic store structures both within and across arenas, as further illustrated in the store topology (Section 2.3.3) and static typing (Section 3.3).

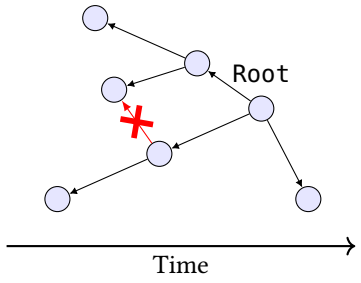
2.3 Store Topology

Languages with static safety guarantees typically impose invariants on the shape of the store to ensure tractable reasoning. These invariants define which forms of sharing, mutation, and lifetime behaviors are possible in permitted programs. Comparing the topological restrictions across systems sheds light on the design trade-offs underlying their safety models.

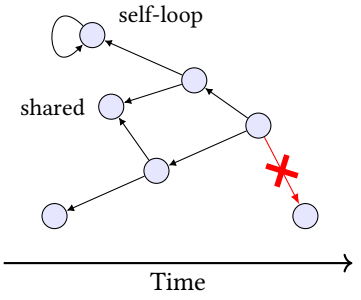
We begin by reviewing Rust as a well-known instance of ownership types, then discuss the prior work on reachability types, and finally introduce the store topology of our work.

2.3.1 Store Topology in Rust. Rust's memory model enforces strict, statically checked ownership, where each heap-allocated object has exactly one unique owner. These ownership relationships induce a tree-shaped object graph (Figure 2a), where each object has at most one inbound edge from its owner. In the store topology, each ownership edge grants exclusive control to read, write, and deallocate the corresponding object.

Rust also supports borrowed references (B& or `&mut`) that grant temporary access without transferring ownership. Borrows neither extend the owner's lifetime nor alter the ownership hierarchy.



(a) The store topology in Rust is a tree structure without *telescoping* over time. The directed edges represent *ownership* of the inbound location where each location is uniquely owned by its parent. Later allocated locations can be owned by earlier allocated locations, but shared ownership is not permitted.



(b) The store topology of reachability types is a directed acyclic graph with self-loops. The directed edges represent reachability from the outbound location to the inbound one. Shared reachability of a location is allowed. *Telescoping* ensures that no location can reach later allocated locations.

Fig. 2. The comparison of store topology between Rust [Klabnik and Nichols 2019] (left) and reachability type [Deng et al. 2025a] (right). Circular nodes in purple represent store locations. Directed edges represent the references from one location to another, ruled by either ownership or reachability, respectively. The timeline indicates the order of allocation in the program flow.

The borrow checker statically ensures that all borrows are strictly bounded by the lifetime of the original owner, and cannot outlive the owner to cause use-after-free or dangling reference errors.

Deallocation of an object is triggered automatically when the unique owner goes out of scope, with all associated borrows guaranteed to have ended beforehand. This tree-shaped discipline yields predictable, statically enforced deallocation, but restricts flexible sharing and cyclic structures.

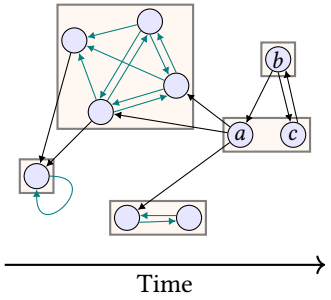
2.3.2 Store Topology in Reachability Types. The store topology in reachability types is more permissive than Rust's, as objects may have multiple inbound edges enabling shared access, as shown in Figure 2b. However, reachability types have their own limitation that telescoping structures (Section 2.2) prevent cyclic store structures.

While reachability types enable static reasoning about separation and aliasing, they do not provide an integrated treatment of deallocation. Existing systems assume that unreachable resources are eventually reclaimed by garbage collection. As potential extensions, prior work [Bao et al. 2021; Wei et al. 2024] has informally proposed, but not implemented, flow-sensitive effect systems to reason about safe deallocation.

2.3.3 Topology in Our Work. As shown in Figure 3a, our system generalizes the topologies by supporting cross-arena sharing and allowing arbitrary internal structures within each arena. Unlike earlier reachability systems that track dependencies at the granularity of individual references, we employ a *coarse-grained* (Section 3.2) model that tracks reachability collectively at the arena level.

All objects in the same arena share a common reachability identity, which eliminates telescoping constraints and enables arbitrary intra-arena object graphs and cyclic structures across arenas (Figure 3b). This relaxation preserves reasoning about separation and safety while substantially expanding expressiveness.

Our system further introduces sound, bulk, and guaranteed deallocation through scoped allocation. Unlike Rust, which statically enforces that no reference outlives its owner, our system permits



(a) The store topology of this work. Internal reachability (teal) is unrestricted within arenas and allows arbitrary structures. Cross-arena reachability (black) can be forward in time to subsequently coallocated locations, supporting cross-arena cycles.

```
1  val a = new Ref(...)
2        // : Top^a
3  val b = new Ref(a)
4        // : Ref[Top^a]^b
5  val c = new Ref(b) at a
6        // : Ref[Top^b]^c <: Top^a
7  b := c
```

(b) The code illustrates a cross-arena cycle between nodes b and c (left). Type details are eliminated via a top type. Reference b to a collectively reaches all locations in a, which allows c, the reference to b, to be assigned to b, forming a cross-arena cycle.

Fig. 3. The store topology of this work (left) and illustration of cross-arena cycles (right). Arenas are marked as boxes containing multiple store locations filled with light orange. Internal reachability is marked in teal. Telescoping structures and exclusivity are relaxed in our work.

references to deallocated objects to persist. Inbound edges to deallocated arenas are not cut off in the store topology, permitting references to retain reachability to objects that are no longer alive. Static typing ensures that dereferencing these references cannot access deallocated resources, thereby maintaining memory safety under flexible lifetime control.

3 Informal Overview

This section informally introduces the $A^{\blacklozenge}_{<:}$-calculus, which features non-lexical arenas (Section 3.1), and the $\{A\}^{\blacklozenge}_{<:}$-calculus, which extends $A^{\blacklozenge}_{<:}$ with scoped resource management and deallocation reasoning (Section 3.4).

3.1 Arenas: Shadow and Non-Lexical

Starting from $A^{\blacklozenge}_{<:}$, we focus on non-lexical shadow arenas and coarse-grained reachability tracking.

3.1.1 Shadow Arenas. We introduce the notion of *shadow arenas*, describing that arenas in our system have no explicit names or constructors. Instead, they are implicitly identified through references in the surface syntax. Arenas are thus not real citizens in our language, but rather as abstract collections of references. A dummy reference can serve as a sufficient proxy to be passed around explicitly when only the arena is required, such as for coallocation.

A new reference allocation (`new Ref`) implicitly establishes a fresh shadow arena containing only that reference. Subsequent coallocations of form (`new Ref(..) at r`) place new references in the same shadow arena as the one of the *proxy* reference `r`, even though the shadow arena itself is never explicitly named.

As summarized in Table 2, multiple allocation forms interact differently with the shadow arenas, yet the references allocated have a unified reference type. The different outermost reachability qualifiers implicitly encode the behavioral differences between fresh allocation and coallocation.

```
val a0 = new Ref(7)            // : Ref[Int]^a0 ⊣ [ a0 : Ref[Int]^♦ ]
// fresh arena containing {a0}
val a1 = new Ref(8) at a0      // : Ref[Int]^a1 ⊣ [ a1 : Ref[Int]^a0, a0 : Ref[Int]^♦ ]
```

Table 2. Comparison of different reference allocation forms in our system, and how they interact with arenas and lifetime control. Only a qualifier but no type distinction exists among the three introduction forms.

Form	Syntax	Formal	Ref Type	Arena	Lifetime
(fresh) allocation	`new Ref(t)`	ref t	`Ref[T]`$^{\blacklozenge}$	fresh	non-lexical
coallocation	`new Ref(t) at r`	ref t at r	`Ref[T]`^{r}	same as `r`	same as `r`
scoped allocation	`new Ref(t) scoped`	ref t as _ in b	`Ref[T]`$^{\blacklozenge}$	fresh	lexical in b

```
// arena containing {a1,a0}
val a2 = new Ref(9) at a1       // : Ref[Int]^a2 <: Ref[Int]^a1 <: Ref[Int]^a0
// arena containing {a2,a1,a0}
```

The fresh allocation of `a0` declares a fresh arena of a single reference cell, and thus the reference `a0` is marked fresh in the typing context. Coallocating `a1` **`at`** `a0` yields a reference that inherits the reachability qualifier of its proxy reference `a0`, since both of them will reside in the same shadow arena. Hence, `a0` and `a1` are equivalent identifiers for that arena; coallocating `a2` **`at`** either reduces to the same location at the runtime.

3.2 Coarse-grained Reachability Tracking

3.2.1 Two-Dimensional Stores and Bulk Reasoning. Recap that in the previous reachability systems, a dynamic store location ℓ evaluated from a reference can be qualified with ℓ itself, yielding *location individuality* that qualifiers of all store locations are strictly disjoint from each other. Thus, the one-dimensional store model is unsuitable for collective storage or bulk reasoning.

We generalize the store model to two-dimensional, indexing each reference by both its arena *location* and an intra-arena *offset*, denoted conventionally as (ℓ, o). Reachability is tracked coarsely at the arena level (through the location ℓ).

```
val a0 = new Ref(7)             // reduce to (ℓ,o1) : Ref[Int]^ℓ for some fresh ℓ, o1
val a1 = new Ref(8) at a0       // reduce to (ℓ,o2) : Ref[Int]^ℓ for some fresh o2
val a2 = new Ref(9) at a1       // reduce to (ℓ,o3) : Ref[Int]^ℓ for some fresh o3
```

Although the indexes of reduced reference cells differ in their offsets, they share the same reachability ℓ, as they are collectively located in the same arena, allowing bulk reasoning.

3.2.2 Connecting Static and Dynamic Typing. The static typing rule, which assigns a coallocated reference the same qualifier as its proxy reference (prefixed by **`at`**), reflects the coarse-grained view. Consider a partially reduced program segment,

```
// after reducing a0 to (ℓ,o1) : Ref[Int]^ℓ
val a1 = new Ref(8) at (ℓ,o1)   // : Ref[Int]^a1 ⊣ [ a1 : Ref[Int]^ℓ ]
a1                              // : Ref[Int]^ℓ through subtyping
```

Statically, coallocated references are treated as aliases of their proxy references, since they reduce to cells in the same arena, while static typing of a reference approximates the entire arena coarsely. Such dependency in reachability yields two key features of shadow arenas: (1) any reference can serve as a proxy for its arena; and (2) reasoning about a reference implicitly applies to all objects within its arena.

3.2.3 Escaping of Non-lexical Arenas. A notable advantage of shadow arenas is to disentangle the reasoning about *escaping* references from reasoning about their arenas. References can safely outlive the lexical scope in which they are allocated, without requiring arenas to be passed or named explicitly. When captured by closures, both references and arenas become *non-lexical*, yet the type system should preserve their tracking for safety.

```
def c1() = {
  val a = new Ref(())
  new Ref(7) at a
} // : (Unit => Ref[Int]♦)
c1() // : Ref[Int]♦
```

```
val c2 = {
  val b = new Ref(())
  () => { new Ref(8) at b }
} // : (Unit => Ref[Int]^c2)
c2() // : Ref[Int]^c2
```

Fig. 4. Comparison between escaping of a reference in a fresh arena (left) and an existing arena (right).

To illustrate the challenges of typing non-lexical arenas, consider the example of two closures wrapping different coallocation patterns as shown in Figure 4. Each invocation of `c1` allocates a fresh arena, while the closure `c2` captures an existing arena. The type system must distinguish the two patterns. In our design, the reachability qualifier of a coallocated reference inherits that of its proxy (reflecting the entire arena); reasoning about escaping coallocated references reduces to the standard reachability reasoning about escaping references (Section 2.1).

3.3 Relaxation of Telescoping Structures

Coarse-grained reachability tracking also relaxes the telescoping structures in prior reachability systems, permitting multi-hop cycles through the store. The coarse-grained reachability allows tracking future resources not yet coallocated in an arena without enlarging the reachability qualifier, as the entire arena shares the same reachability.

We illustrate the relaxation by showcasing the mutually recursive example, which is not allowed in previous reachability type systems, as discussed in Section 2.2.

```
val a = new Ref(())              // : Ref[Unit]^a
val c1 = ... at a                // : Ref[(Int => Int)^a]^c1 <: Ref[(Int => Int)^a]^a
val c2 = ... at a                // : Ref[(Int => Int)^a]^c2 <: Ref[(Int => Int)^a]^a
c1 := (x : Int) => (!c1)(x)      // OK. RHS : (Int => Int)^c1 <: (Int => Int)^a
c2 := (x : Int) => (!c2)(x)      // OK.
```

By coallocating `c1` and `c2` in the same arena `a`, they can reach each other via the reachability to the entire arena. More generally, arenas support arbitrary internal store structures, including self-loops and cyclic store structures. Our system supports sound reasoning about such cyclic structures as a whole, such as separation and safe deallocation (Section 6.3).

Using the same function capturing techniques, cross-arena multi-hop cycles can also be encoded, as illustrated in Figure 3b.

3.4 Scopes: Bulk and Guaranteed Deallocation

While reachability types reason about aliasing and separation, they provide no mechanism for user-controlled, timely resource reclamation. The $\{A\}^{\blacklozenge}_{<:}$-calculus extends $A^{\blacklozenge}_{<:}$ with scoped resource management, supporting automatic and bulk deallocation tied to lexical scopes. Importantly, these deallocation guarantees are lightweight: they require no explicit effect tracking and are enforced solely upon the surface-level **scoped** annotation.

3.4.1 Guaranteed Deallocation. To regain user control of lifetime, we extend the $A^{\blacklozenge}_{<:}$-calculus with *scoped* allocation, whose lifetimes are bounded by their lexical blocks.

```
{
  val a = new Ref(()) scoped     // : Ref[Unit]^a
  !a                             // : Unit
} // {a} is deallocated
```

The **scoped** annotation marks the reference `a` and subsequent references coallocated **at** `a` for automatic deallocation at the end of the block. Specifically, as summarized in Table 2, scoped allocation produces references having no type distinction from those created by fresh allocation, behaving no

Syntax $\mathsf{A}^{\blacklozenge}_{<:}$

x, y, z	$\in$	Var	Variables
f, g, h	$\in$	Var	Function Variables
X	$\in$	Var	Type Variables
t	$::=$	$c \mid x \mid \lambda f(x).t \mid t\ t \mid \mathsf{ref}\ t \mid \mathsf{ref}\ t\ \mathsf{at}\ t \mid !\,t \mid t := t$ $\mid \Lambda f(X^{x}).t \mid t\ [Q\,]$	Terms
p, q, r, w	$\in$	$\mathcal{P}_{\mathsf{fin}}(\mathsf{Var} \uplus \{\blacklozenge\})$	Reachability Qualifiers
S, T, U, V	$::=$	$B \mid f(x : Q) \to Q \mid \mathsf{Ref}\ [Q]$ $\mid \mathsf{Top} \mid \forall f(X^{x} <: Q).Q$	Types
O, P, Q, R	$::=$	T^{q}	Qualified Types
φ	$\in$	$\mathcal{P}_{\mathsf{fin}}(\mathsf{Var})$	Observations
Γ	$::=$	$\varnothing \mid \Gamma, x : Q \mid \Gamma, X^{x} <: Q$	Typing Environments

Qualifier Notations

$$p, q := p \cup q \qquad x := \{x\} \qquad \blacklozenge := \{\blacklozenge\} \qquad \blacklozenge q := \{\blacklozenge\} \cup q$$

Fig. 5. The syntax of $\mathsf{A}^{\blacklozenge}_{<:}$. Extensions from Wei et al. [2024] and Deng et al. [2025a] are shaded.

differently within their scope. The type system ensures that such references cannot escape their defined scopes, preserving memory safety and predictable reclamation.

3.4.2 Bulk Deallocation. When a scoped reference is deallocated, all resources coallocated within its arena are released together.

```
val sc2 = {
  val a = new Ref(()) scoped    // : Ref[Unit]^a
  val u = new Ref(42) at a      // : Ref[Int]^u
  val v = new Ref(!u)           // : Ref[Int]^v
  new Ref(v)                    // : Ref[Ref[Int]^v]^◆
} // {a,u} are deallocated
!sc2                            // OK
```

To ensure static safety, the type system prohibits scoped references in either the type or the qualifier of the scope return, thereby preventing any resource in the scoped arena from leaking.

Scopes may nest to form a stack discipline that mirrors classic region-based resource management schemes. At the meta level, the extent of each scope can be derived from the placement of scoped references, including those in function calls, and we sketch such a translation in Section 5.2.

4 $\mathsf{A}^{\blacklozenge}_{<:}$-Calculus: Arena-Based Reachability Tracking

We present the $\mathsf{A}^{\blacklozenge}_{<:}$-calculus, an extension of the $\mathsf{F}^{\blacklozenge}_{<:}$-calculus [Wei et al. 2024] with shadow arenas, coarse-grained reachability tracking, and two-dimensional store semantics. In this section, we primarily focus on the static typing, the call-by-value operational semantics, and the store model. Dynamic typing is deferred to Section 5.

4.1 Syntax

Figure 5 defines the formal syntax of $\mathsf{A}^{\blacklozenge}_{<:}$. It follows the conventions of $\mathsf{F}^{\blacklozenge}_{<:}$ [Wei et al. 2024] for continuity. Extensions related to shadow arenas are specifically marked to highlight.

Qualifier Substitution $q[p/x]$ $q[p/\blacklozenge]$

$$q[p/x] = q \setminus \{x\} \cup p \quad x \in q \qquad q[p/x] = q \quad x \notin q$$
$$q[p/\blacklozenge] = q \cup p \quad \blacklozenge \in q \qquad q[p/\blacklozenge] = q \quad \blacklozenge \notin q$$

Transitive Closure and Overlap $\Gamma \vdash q^n$ $\Gamma \vdash p \between q$

Transitive Closure $\Gamma \vdash q^0 := q$ $\qquad \Gamma \vdash q^{n+1} := ((\bigcup_{x \in q} \{ y \mid x : T^{d,y} \in \Gamma \}) \cup q)^n$

Qualifier Overlap $\Gamma \vdash p \between q := \blacklozenge(p^{|\Gamma|} \cap q^{|\Gamma|})$

Fig. 6. Computable transitive closure and overlap of qualifiers in $A^{\blacklozenge}_{<:}$.

Terms include value constants, variables, λ-abstractions, applications, reference operations, and System $F_{<:}$ type abstractions and instantiations. Function and type abstractions are explicitly annotated with their self-references to allow their bodies to observe their own reachability.

In addition to standard reference allocation, we introduce *coallocation* of form ref t_1 at t_2 (surface syntax `new Ref(t1) at t2`), which allocates a reference within the arena hosting t_2. Reference operations are uniform across allocation forms, including the scoped allocation form later.

Qualified types T^q are types annotated with reachability qualifiers q, which are finite sets of variables (and later, store locations), and optionally with a freshness marker $\blacklozenge$. We use φ to denote an observation filter, a special qualifier specifying the upper bound of resources visible to the typing judgment. Γ denotes the typing context, and later Σ denotes the store typing.

The $F_{<:}$-style quantification $X^x <: Q$ introduces a type variable X and a qualifier variable x that can be used separately. The syntax is an abbreviation for simplicity since types and qualifiers are often quantified together, *i.e.*, $\forall(X <: T).\forall(x <: q).Q \equiv \forall(X^x <: T^q).Q$.

Figure 6 defines qualifier substitution, transitive closure, and overlap. *Transitive closure* computes the maximal qualifier reachable from variables in the original qualifier by recursively unfolding variables to the associated qualifiers in Γ. Because qualifiers are assigned minimally for variables, two syntactically disjoint qualifiers may nevertheless refer to shared store locations via aliasing. Qualifier overlap is therefore defined over transitive closures rather than set intersection.

4.2 Static Typing and Subtyping

Figure 7 and Figure 8 present the declarative typing and subtyping rules of $A^{\blacklozenge}_{<:}$. All references share a single reference type, independent of their allocation form. Unlike Deng et al. [2025a], who introduce specialized cyclic reference types with constraints, our system retains full deep dependencies of arguments while supporting cycles, which will be discussed in Section 6.1. Shadow arenas do not show up in static typing, simplifying the surface syntax. In the absence of coallocation (T-REFAT), $A^{\blacklozenge}_{<:}$ collapses to the declarative typing system of Wei et al. [2024], since coarse-grained reachability tracking degenerates to fine-grained tracking over arenas of size one.

Subtyping. Subtyping is defined over qualifiers (Q-), types (S-), and qualified types (SQ-SUB). Type-level subtyping rules resemble those of System $F_{<:}$ and concern qualified types. Qualifier subtyping includes context-independent rules (Q-SUB) and (Q-CONG) that structurally enlarge qualifiers, and context-dependent rules (Q-VAR) and (Q-SELF). Variables may unfold to their recorded qualifiers (Q-VAR), and function self-references may additionally fold their captured resources (Q-SELF).

Term Typing. Term typing judgments are regulated by an observation filter φ, which bounds maximal observable resources during typing. The variable typing rule (T-VAR) assigns the variable itself as the minimal qualifier. Function abstraction (T-ABS) types the body under its function

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

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

$$\frac{\Gamma^{\varphi} \vdash t : T^{q} \qquad \blacklozenge \notin q}{\Gamma^{\varphi} \vdash \mathsf{ref}\ t : (\mathsf{Ref}\ [T^{q}])^{\blacklozenge}}\ (\textsc{t-ref}) \qquad \frac{\Gamma^{\varphi} \vdash t_1 : T^{q} \qquad \blacklozenge \notin q \quad \Gamma^{\varphi} \vdash t_2 : \mathsf{Ref}\ [U]^{p}}{\Gamma^{\varphi} \vdash \mathsf{ref}\ t_1\ \mathsf{at}\ t_2 : (\mathsf{Ref}\ [T^{q}])^{p}}\ (\textsc{t-refat})$$

$$\frac{\Gamma^{\varphi} \vdash t : \mathsf{Ref}\ [T^{p}]^{q} \quad \blacklozenge \notin p \qquad p \subseteq \varphi}{\Gamma^{\varphi} \vdash\ !t : T^{p}}\ (\textsc{t-deref}) \qquad \frac{\Gamma^{\varphi} \vdash t_1 : \mathsf{Ref}\ [T^{p}]^{q} \quad \Gamma^{\varphi} \vdash t_2 : T^{p} \quad \blacklozenge \notin p}{\Gamma^{\varphi} \vdash t_1 \coloneqq t_2 : \mathsf{Unit}^{\varnothing}}\ (\textsc{t-assgn})$$

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

$$\frac{\Gamma^{\varphi} \vdash t_1 : (f(x : T^{p}) \to Q)^{q} \quad \Gamma^{\varphi} \vdash t_2 : T^{p} \qquad \blacklozenge \notin p \qquad f \notin \mathrm{fv}(U) \quad Q = U^{r} \qquad r \subseteq \blacklozenge\varphi, x, f}{\Gamma^{\varphi} \vdash t_1\ t_2 : Q[p/x, q/f]}\ (\textsc{t-app}) \qquad \frac{\Gamma^{\varphi} \vdash t : (\forall f(X^{x} <: T^{p}).Q)^{q} \quad p \subseteq \varphi \qquad \blacklozenge \notin p \qquad f \notin \mathrm{fv}(U) \quad Q = U^{r} \qquad r \subseteq \blacklozenge\varphi, x, f}{\Gamma^{\varphi} \vdash t[T^{p}] : Q[T^{p}/X^{x}, q/f]}\ (\textsc{t-tapp})$$

$$\frac{\Gamma^{\varphi} \vdash t_1 : \left(f(x : T^{p \curvearrowright q}) \to Q\right)^{q} \quad \Gamma^{\varphi} \vdash t_2 : T^{p} \qquad \blacklozenge \in p \Rightarrow x \notin \mathrm{fv}(U) \quad Q = U^{r} \qquad r \subseteq \blacklozenge\varphi, x, f \qquad f \notin \mathrm{fv}(U)}{\Gamma^{\varphi} \vdash t_1\ t_2 : Q[p/x, q/f]}\ (\textsc{t-app}^{\blacklozenge}) \qquad \frac{\Gamma^{\varphi} \vdash t : \left(\forall f(X^{x} <: T^{p \curvearrowright q}).Q\right)^{q} \quad p \subseteq \varphi \qquad \blacklozenge \in p \Rightarrow x \notin \mathrm{fv}(U) \quad Q = U^{r} \qquad r \subseteq \blacklozenge\varphi, x, f \qquad f \notin \mathrm{fv}(U)}{\Gamma^{\varphi} \vdash t[T^{p}] : Q[T^{p}/X^{x}, q/f]}\ (\textsc{t-tapp}^{\blacklozenge})$$

Fig. 7. Typing rules of $\mathsf{A}^{\blacklozenge}_{<:}$. Extensions from Wei et al. [2024] and Deng et al. [2025a] are shaded.

qualifier, argument, and self-reference as the observation filter, ensuring all the resources required for function computation are captured in the function qualifier.

Application is divided into two cases: precise application (T-APP) and growing application (T-APP$^{\blacklozenge}$). Precise application requires the argument qualifier to be fully contained in the function's domain qualifier. Growing application, with a freshness marker $\blacklozenge$ in the function domain qualifier, permits the argument to reach contextually fresh resources beyond the function qualifier. The function domain qualifier in the growing application is overridden into the upper bound of *controlled sharing*, requiring that the overlap between the function qualifier and the argument qualifier $(p \curvearrowright q)$ lies within that bound. Our application adheres to the same constraints in Wei et al. [2024].

The function codomain may depend on function and argument qualifiers by allowing the *deep occurrences* of the bound variables f and x in the return type and qualifier, which supports lightweight reachability polymorphism [Wei et al. 2024]. $\mathsf{F}_{<:}$-style polymorphism is encoded by leveraging the reachability polymorphism. The universal instantiation is treated as application to a phantom argument serving as the type witness, so that rules (T-TABS), (T-TAPP), and (T-TAPP$^{\blacklozenge}$) are structurally similar to function abstraction and applications.

Reference Typing. As references introduced in different forms share a single reference type, reference assignment (T-ASSGN) and dereferencing (T-DEREF) are generic. Fresh allocation (T-REF) introduces a new reference marked fresh, corresponding to standard allocation in Wei et al. [2024]. The referent qualifier must be non-fresh to retain sound reachability tracking; otherwise, the same fresh referent could be bound to multiple non-aliasing variables, leading to unsoundness:

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

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash p <: q}{\Gamma \vdash S^p <: T^q} \text{(SQ-SUB)}$$

$$\frac{}{\Gamma \vdash T <: T} \text{(S-REFL)} \qquad \frac{\Gamma \vdash T <: S \qquad \Gamma \vdash S <: U}{\Gamma \vdash T <: U} \text{(S-TRANS)} \qquad \frac{\Gamma \vdash p <: q \qquad \Gamma \vdash q <: r}{\Gamma \vdash p <: r} \text{(Q-TRANS)}$$

$$\frac{}{\Gamma \vdash T <: \mathsf{Top}} \text{(S-TOP)} \qquad \frac{p \subseteq q \subseteq \blacklozenge dom(\Gamma)}{\Gamma \vdash p <: q} \text{(Q-SUB)}$$

$$\frac{X^x <: T^q \in \Gamma}{\Gamma \vdash X <: T} \text{(S-TVAR)} \qquad \frac{\Gamma \vdash q_1 <: q_2}{\Gamma \vdash p, q_1 <: p, q_2} \text{(Q-CONG)}$$

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: S \qquad q \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathsf{Ref}\ S^q <: \mathsf{Ref}\ T^q} \text{(S-REF)} \qquad \frac{f : T^q \in \Gamma \qquad \blacklozenge \notin q}{\Gamma \vdash q, f <: f} \text{(Q-SELF)}$$

$$\frac{\Gamma \vdash P <: O \qquad \Gamma, f : (f(x : O) \to Q)^{\blacklozenge}, x : P \vdash Q <: R}{\Gamma \vdash f(x : O) \to Q <: f(x : P) \to R} \text{(S-FUN)} \qquad \frac{x : T^q \in \Gamma \qquad \blacklozenge \notin q}{\Gamma \vdash x <: q} \text{(Q-VAR)}$$

$$\frac{\Gamma \vdash P <: O \qquad \Gamma, f : (\forall f(X^x <: O).Q)^{\blacklozenge}, x : X^x <: P \vdash Q <: R}{\Gamma \vdash \forall f(X^x <: O).Q <: \forall f(X^x <: P).R} \text{(S-ALL)} \qquad \frac{X^x <: T^q \in \Gamma \qquad \blacklozenge \notin q}{\Gamma \vdash x <: q} \text{(Q-QVAR)}$$

Fig. 8. Subtyping rules of $\mathsf{A}^{\blacklozenge}_{<:}$.

```
val r  = new Ref(new Ref(42))   // : Ref[Ref[Int]♦]^r
val v1 = !r                     // : Ref[Int]^v1
val v2 = !r                     // : Ref[Int]^v2
free(v1); use(v2)               // Type check but actually use-after-free!
```

While the inner qualifier represents the referent reachability, the outer qualifier tracks only the immediate reference reachability, as *shallow reference tracking* [Deng et al. 2025a].

The freshness marker ♦ in the outer qualifier (T-REF) conceptually represents a fresh shadow arena with the fresh allocation. Subsequent coallocation (T-REFAT) places a new reference into an existing arena by assigning it the same outer qualifier as the proxy reference in the `at` clause, as discussed in Section 3.2. The proxy reference itself may be fresh, *e.g.*, `new Ref(42) at new Ref(7)` safely identifies an unnamed arena, since the proxy becomes unreachable afterwards.

4.3 Dynamic Semantics and Two-Dimensional Stores

The $\mathsf{A}^{\blacklozenge}_{<:}$-calculus generalizes the small-step call-by-value semantics of $\mathsf{F}^{\blacklozenge}_{<:}$ to a two-dimensional store supporting arena-based memory management. Figure 9 summarizes the operational semantics.

4.3.1 Two-Dimensional Stores. As illustrated in Figure 1b, our two-dimensional store follows the traditional arena-based layouts, mapping from *store indexes*, pairs of *locations* and *offsets* (ℓ, o), to reference cells. An arena at location ℓ conceptually describes the collection of reference cells sharing the same location ℓ and differing only in offsets. The layout is described as *two-dimensional* to emphasize that references are grouped by arenas, while no separation or hierarchy constraints

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

$$\ell \in \mathsf{Loc} \qquad o \in \mathsf{Offset} \qquad \Sigma ::= \varnothing \mid \Sigma,\ (\ell, o) : Q \qquad \mathrm{dom}(\Sigma) := \{(\ell, o) \mid (\ell, o) \in \Sigma\}$$
$$p, q, r \subseteq \mathcal{P}_{\mathsf{fin}}(\mathsf{Var} \uplus \mathsf{Loc} \uplus \{\blacklozenge\}) \qquad \varphi \subseteq \mathcal{P}_{\mathsf{fin}}(\mathsf{Var} \uplus \mathsf{Loc}) \qquad \mathrm{dom}\ell(\Sigma) := \{\ell \mid (\ell, o) \in \Sigma\}$$

$$\frac{\Sigma(\ell, o) = T^{\,q} \quad q \subseteq \mathrm{dom}\ell(\Sigma) \quad \mathrm{fv}(T) = \varnothing \quad \mathrm{ftv}(T) = \varnothing \quad \ell \in \varphi}{[\Gamma \mid \Sigma]^{\varphi} \vdash (\ell, o) : \mathsf{Ref}\ [T^{\,q}]^{\ell}} \ (\textsc{t-loc})$$

Well-Formed and Well-Typed Stores $\boxed{\Sigma\ \mathsf{ok}}$ $\boxed{[\Gamma \mid \Sigma]^{\varphi} \vdash \sigma}$

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

$$\frac{}{\varnothing\ \mathsf{ok}} \qquad \frac{\Sigma\ \mathsf{ok} \quad \mathrm{fv}(T) = \varnothing \quad \mathrm{ftv}(T) = \varnothing \quad q \subseteq \mathrm{dom}\ell(\Sigma) \quad o > 0 \Rightarrow \ell \in \mathrm{dom}\ell(\Sigma)}{\Sigma,\ (\ell, o) : T^{\,q}\ \mathsf{ok}}$$

Reduction Contexts, Values, Terms, Stores

$$C ::= \square \mid C\ t \mid v\ C \mid \mathsf{ref}\ C \mid \mathsf{ref}\ t\ \mathsf{at}\ C \mid \mathsf{ref}\ C\ \mathsf{at}\ v \mid\ !C \mid C := t \mid v := C \mid C\ [Q]$$
$$v ::= \lambda f(x).t \mid c \mid (\ell, o) \mid \mathsf{unit} \mid \Lambda f(X^{x}).t \qquad t ::= \cdots \mid (\ell, o) \qquad \sigma ::= \varnothing \mid \sigma,\ (\ell, o) \mapsto v$$

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

$$C[\ (\lambda f(x).t)\ v\ ] \mid \sigma \longrightarrow C[\ t[v/x, (\lambda f(x).t)/f]\ ] \mid \sigma \qquad (\beta)$$
$$C[\ \mathsf{ref}\ v\ ] \mid \sigma \longrightarrow C[\ (\ell, 0)\ ] \mid (\sigma,\ (\ell, 0) \mapsto v) \qquad (\ell, 0) \notin \mathrm{dom}(\sigma) \qquad (\textsc{ref})$$
$$C[\ \mathsf{ref}\ v\ \mathsf{at}\ (\ell, o)\ ] \mid \sigma \longrightarrow C[\ (\ell, o')\ ] \mid (\sigma,\ (\ell, o') \mapsto v) \qquad (\ell, o') \notin \mathrm{dom}(\sigma) \qquad (\textsc{refat})$$
$$C[\ !(\ell, o)\ ] \mid \sigma \longrightarrow C[\ \sigma((\ell, o))\ ] \mid \sigma \qquad (\ell, o) \in \mathrm{dom}(\sigma) \qquad (\textsc{deref})$$
$$C[\ (\ell, o) := v\ ] \mid \sigma \longrightarrow C[\ \mathsf{unit}\ ] \mid \sigma[(\ell, o) \mapsto v] \qquad (\ell, o) \in \mathrm{dom}(\sigma) \qquad (\textsc{assign})$$
$$C[\ (\Lambda f(X^{x}).t)\ Q\ ] \mid \sigma \longrightarrow C[\ t[Q/X^{x}, (\Lambda f(X^{x}).t)/f]\ ] \mid \sigma \qquad (\beta_T)$$

Fig. 9. Store typings and call-by-value reduction of $\mathsf{A}^{\blacklozenge}_{<:}$ (Section 4). Extensions from Wei et al. [2024] and Deng et al. [2025a] are shaded.

are imposed across arenas. References may freely reach other arenas regardless of allocation order, in contrast to traditional region-based systems with explicit regions (Section 2.3.3).

4.3.2 Reduction. The $\mathsf{A}^{\blacklozenge}_{<:}$-calculus generalizes the call-by-value reduction of $\mathsf{F}^{\blacklozenge}_{<:}$ to populate the two-dimensional store. Fresh allocation and coallocation extend the two-dimensional store in either rows or columns, respectively. Reducing a coallocation, ref t_1 at t_2, first evaluates the proxy reference t_2 to a concrete store index, and the new reference cell is then placed at a fresh offset within that location after the evaluation of t_1. We omit the dynamic typing, as a subset of Figure 12.

Once reduced to a precise store index, a reference is tracked solely by its location ℓ, equivalently the arena location. Thus, during reduction, the freshness marker $\blacklozenge$ in the qualifier is substituted with its location ℓ, which is reflected in the preservation theorem.

4.4 Metatheory

We state the key soundness theorems of $\mathsf{A}^{\blacklozenge}_{<:}$. Proof details are omitted as $\mathsf{A}^{\blacklozenge}_{<:}$-calculus is a strict subset of the $\{\mathsf{A}\}^{\blacklozenge}_{<:}$-calculus (Section 5.5).

Theorem 4.1 (Progress). *If* $[\varnothing \mid \Sigma]^{\varphi} \vdash t : Q$ *and* $\Sigma\ \mathsf{ok}$, *then either* t *is a value, or for any store* σ *where* $[\varnothing \mid \Sigma]^{\varphi} \vdash \sigma$, *there exists a term* t' *and a store* σ' *such that* $t \mid \sigma \longrightarrow t' \mid \sigma'$.

Term Syntax, Reduction Contexts $\boxed{\{\mathsf{A}\}^{\blacklozenge}_{<:}}$

$$t ::= \cdots \mid \mathsf{ref}\ t\ \mathsf{as}\ x\ \mathsf{in}\ t \mid \mathsf{local}\ \ell\ \mathsf{in}\ t \qquad C ::= \cdots \mid \mathsf{ref}\ C\ \mathsf{as}\ x\ \mathsf{in}\ t \mid \mathsf{local}\ \ell\ \mathsf{in}\ C$$

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

$$C[\ \mathsf{ref}\ v\ \mathsf{as}\ x\ \mathsf{in}\ t\ ] \mid \sigma \longrightarrow C[\ \mathsf{local}\ \ell\ \mathsf{in}\ t[(\ell,0)/x]\ ] \mid (\sigma, (\ell,0) \mapsto v) \quad (\ell,0) \notin \mathrm{dom}(\sigma) \quad (\textsc{with})$$
$$C[\ \mathsf{local}\ \ell\ \mathsf{in}\ v\ ] \mid \sigma \longrightarrow C[\ v\ ] \mid \sigma[(\ell,*) \mapsto \mathrm{None}] \quad (\textsc{close})$$

Static Term Typing $\boxed{\Gamma \vdash t : Q}$

$$\frac{\Gamma^{\varphi} \vdash t_1 : T^{p} \quad (\Gamma\ ,\ x : \mathsf{Ref}\ [T^{p}]^{\blacklozenge})^{\varphi, x} \vdash t_2 : Q \quad \blacklozenge \notin p \quad x \notin \mathrm{fv}(Q)}{\Gamma^{\varphi} \vdash \mathsf{ref}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : Q} \quad (\textsc{t-refin})$$

Fig. 10. Extensions of $\{\mathsf{A}\}^{\blacklozenge}_{<:}$ on $\mathsf{A}^{\blacklozenge}_{<:}$ in syntax, static typing, and reduction. The term local ℓ in t (marked in violet) is hidden from the surface syntax.

Theorem 4.2 (Preservation). *If* $[\varnothing \mid \Sigma]^{\varphi} \vdash t : T^{q}$, *and* $[\varnothing \mid \Sigma]^{\varphi} \vdash \sigma$, *and* Σ ok, *and* $t \mid \sigma \longrightarrow t' \mid \sigma'$, *then there exists* $\Sigma' \supseteq \Sigma, \varphi' \supseteq \varphi \cup p$, *and* $p \supseteq \mathrm{dom}\ell(\Sigma'/\Sigma)$ *such that* $[\varnothing \mid \Sigma']^{\varphi'} \vdash \sigma'$ *and* $[\varnothing \mid \Sigma']^{\varphi'} \vdash t' : T^{q[p/\blacklozenge]}$.

These theorems have been mechanized in Rocq.

5 $\{\mathsf{A}\}^{\blacklozenge}_{<:}$-Calculus: Scoped Resource Management

This section presents the $\{\mathsf{A}\}^{\blacklozenge}_{<:}$-calculus, which extends the $\mathsf{A}^{\blacklozenge}_{<:}$-calculus with scoped resource management and deallocation reasoning. Since our deallocation reasoning requires no type distinctions or surface level annotations, most static typing rules remain unchanged from $\mathsf{A}^{\blacklozenge}_{<:}$-calculus. Content specific to dynamic typing is highlighted in violet, while removing these dynamic constraints yields the corresponding static system.

5.1 Language Extensions on $\mathsf{A}^{\blacklozenge}_{<:}$

Figure 10 summarizes the language extensions from $\mathsf{A}^{\blacklozenge}_{<:}$-calculus to $\{\mathsf{A}\}^{\blacklozenge}_{<:}$-calculus. The term syntax is extended with the static scope introduction form, ref t_1 as x in t_2, and its dynamic elimination form, local ℓ in t. The introduction form declares a scoped reference of t_1 whose lifetime is lexically bound with x in t_2, while the elimination form is internal and not part of the surface syntax.

Static Typing. The static typing rule (T-REFIN) combines aspects of fresh reference allocation, function abstraction, and application. Intuitively, the introduction form ref t_1 as x in t_2 types the body t_2 in a context extended with a freshly allocated reference x for t_1, analogous to introducing a reference via let-binding. The rule follows the idea of scoped acquisition and release discipline, that resources are acquired at scope entry and released at scope exit. The non-escaping requirement, $x \notin \mathrm{fv}(Q)$, is encoded as deep disjointedness between the scoped reference x and the qualified type Q of the scope return t_2, which ensures no downstream dependencies on x.

Reduction. Reduction of a scope proceeds in two phases. First, after the resource t_1 reduces to a value, a fresh reference (and its underlying arena) is allocated at $(\ell, 0)$ for a fresh ℓ, mirroring fresh allocation. The scope introduction form then reduces to its elimination form local ℓ in t_2, where occurrences of the bound variable x in t_2 are substituted with the concrete store index $(\ell, 0)$. Subsequently, t_2 is evaluated within the elimination form, and once t_2 reduces to a value v, the elimination form deallocates the entire arena at ℓ and unpacks v, thereby closing the scope.

Selected Syntax Translation $\boxed{[\![t_s]\!] = \lambda\kappa.\, t_c}$

$$
\begin{aligned}
[\![\ \texttt{new Ref(t) scoped}\ ]\!] &= \lambda\kappa.\, [\![\ \texttt{t}\ ]\!]\ (\lambda t'.\, \text{ref } t' \text{ as } \underline{x} \text{ in } \kappa[\underline{x}]) && \underline{x} \text{ is fresh}\\
[\![\ \texttt{def f(x)= t}\ ]\!] &= \lambda\kappa.\, \kappa[\lambda f(x).\, [\![\ \texttt{t}\ ]\!]\ (\lambda t'.\, t')]\\
[\![\ \texttt{t1(t2)}\ ]\!] &= \lambda\kappa.\, [\![\ \texttt{t1}\ ]\!]\ (\lambda t_1'.\, [\![\ \texttt{t2}\ ]\!]\ (\lambda t_2'.\, \kappa[t_1'\ t_2']))\\
[\![\ \texttt{x}\ ]\!] &= \lambda\kappa.\, \kappa[x]
\end{aligned}
$$

Fig. 11. The selected syntax translation from the surface syntax to the core syntax of $\{\mathsf{A}\}^{\blacklozenge}_{<:}$.

5.2 Surface Syntax Translation

Careful readers may notice that the surface syntax of scope introduction (Figure 1a) does not strictly match the core syntax in the formal calculus. In particular, scoped allocation (marked with **scoped**) may appear mid-block, whereas the core calculus expects it at block start. This discrepancy can be resolved by a translation to align the lifetime of scoped references with blocks. Namely, a scoped reference declared within a block is translated so that its lifetime extends to the block end.

A subtle case arises when scoped allocation appears as a function argument, *e.g.*, `f(new Ref(t) scoped)`. Here, the scoped reference must outlive argument evaluation and potentially align with the lifetime of the function return. We handle this by a meta-level translation that lifts the allocation outward and binds it earlier.

Figure 11 sketches the translation from surface syntax (in dark cyan) to the core syntax in the formal calculus (in magenta). The sketch follows the notation convention of Danvy and Filinski [1990], where an abstract continuation κ is a context-dependent meta-level function passed throughout the translation. To initiate a translation, we apply $[\![t_s]\!]$ on an identity continuation $\lambda x.x$.

We showcase translation from the example surface syntax `f(new Ref(v) scoped)` to core calculus:

$$
\begin{aligned}
& [\![\ \texttt{f(new Ref(t) scoped)}\ ]\!]\ (\lambda x.\, x)\\
=\ & \lambda\kappa.\, [\![\ \texttt{f}\ ]\!]\ (\lambda f'.\, [\![\ \texttt{new Ref(t) scoped}\ ]\!]\ (\lambda r'.\, \kappa[f'\ r']))\ (\lambda x.\, x) && \text{application}\\
=\ & (\lambda\kappa.\, \kappa[f])\ (\lambda f'.\, [\![\ \texttt{new Ref(t) scoped}\ ]\!]\ (\lambda r'.\, f'\ r')) && \text{variable } f\\
=\ & (\lambda\kappa.\, [\![\ \texttt{t}\ ]\!]\ (\lambda t'.\, \text{ref } t' \text{ as } \underline{x} \text{ in } \kappa[\underline{x}]))\ (\lambda r'.\, f\ r') && \text{scoped reference}\\
=\ & (\lambda\kappa.\, \kappa[t])\ (\lambda t'.\, \text{ref } t' \text{ as } \underline{x} \text{ in } f\ \underline{x}) && \text{variable } t\\
=\ & \text{ref } t \text{ as } \underline{x} \text{ in } f\ \underline{x}
\end{aligned}
$$

After translation, the function f is applied to the scoped binding $\underline{x}$ within the scope. Other translation rules can be defined straightforwardly and omitted.

5.3 Flow-Insensitive Deallocation Reasoning

The design principle of $\{\mathsf{A}\}^{\blacklozenge}_{<:}$-calculus is to introduce lexical scopes following stack discipline without requiring additional surface annotations or type distinctions. Rather than relying on explicit use-and-kill effects, the calculus reasons about safe deallocation through complementary forms of observation filters and dynamic tracking of locations to be reclaimed. The *flow-insensitive* deallocation reasoning is established by dynamic typing and preserved under reduction without being exposed at the static level. We start from discussing why static non-escaping is sufficient for deallocation safety, and then illustrate how the property is preserved dynamically.

5.3.1 Non-Escaping in Static Typing. The typing rule (T-REFIN) enforces that the scoped variable x does not occur in the type or qualifier of the scope return t_2. This ensures that x is disjoint from downstream computation, since any transitive reachability to x via store lookup would induce a deep occurrence of x in the type:

```
val x = new Ref(..) scoped      // : Ref[T]^x ⊣ [ x : Ref[T]^♦ ], scope body starts
val r1 = x                      // : Ref[T]^r1 <: Ref[T]^x, cannot return r1
```

```
val r2 = new Ref(x)            // : Ref[Ref[T]^x]^r2, cannot return r2
val r3 = new Ref(x)            // : Top^r3, OK to return
val r4 = new Ref(!x)           // : Ref[T]^r4, OK to return
```

Because the scope return must be typed without observing x, internal aliases or references such as `r1` and `r2` are rejected. By contrast, `r3` is permitted after being upcasted to Top, which prevents dereferencing and thus guarantees that x is unreachable. Our system guarantees the reclamation of x itself, but not of its referent, so returning a disjoint reference such as `r4` is allowed.

5.3.2 Non-Trivial Gaps in Dynamics. While the non-escaping property prevents scoped references from escaping their scope, it does not prevent reintroduction of reachability to reclaimed resources during later evaluation. Statically, reintroduction is impossible because the variable binding x is only appended to Γ when typing the body t_2 and not observable otherwise. Dynamically, however, store locations of scoped references can be accessed directly (T-LOC), bypassing the Γ context.

Recall the preservation theorem (Theorem 4.2) inherited from prior reachability type systems [Deng et al. 2025a; Wei et al. 2024]: the observation filter φ monotonically grows to $\varphi \cup p$ where p contains fresh store locations from the reduction. As a result, a reclaimed location could be reintroduced through subsumption (T-SUB) or growing application (T-APP$^\blacklozenge$), unless it is explicitly removed from observation φ. This evidently illustrates the central challenge of deallocation safety.

5.3.3 Observation Filters. To prevent reintroduction of reclaimed locations, deallocated locations must be excluded from the observation filter φ. This entails the invariant that φ remains disjoint from all scoped locations outside their lexical scopes. Accordingly, for a scoped elimination form local ℓ in t_2: (1) ℓ is added to φ when the elimination form is introduced; (2) ℓ remains observable during the evaluation of t_2; and (3) ℓ is removed from φ when the evaluation finishes.

However, shrinking the observation filter φ cannot be arbitrary, as *strengthening* the typing judgment with fewer observable resources is not generally admissible; that is, $[\Gamma \mid \Sigma]^{\varphi} \vdash t : T \not\Longrightarrow [\Gamma \mid \Sigma]^{\varphi \setminus \ell} \vdash t : T$. To address this, observation filters are shrunk only when a scope elimination form reduces to its value (Section 5.3.4), and the dynamic typing ensures that such shrinking does not affect typing outside that scope (Section 5.3.5). This is achieved via a combination of *local locations* and *well-formed stepping* (Figure 14), as well as dynamic typing rules (Figure 12).

5.3.4 Strengthening with Value Witnesses. In reachability types, the qualifier of a *value* characterizes the minimal resources required to type it. For example, the body of a function abstraction (T-ABS) is typed with observation limited to the function qualifier q instead of φ. Formally, qualifiers assigned to well-typed values provide lower bounds for observation in typing.

Lemma 5.1 (Value Observability). *If* $[\Gamma \mid \Sigma]^{\varphi} \vdash v : T^{q}$, *then* $[\Gamma \mid \Sigma]^{q} \vdash v : T^{q}$.

With a value witness, we can shrink the observation filter down to any φ' that contains q.

Lemma 5.2 (Value Retyping). *If* $[\Gamma \mid \Sigma]^{\varphi} \vdash v : T^{q}$, *and* $\ell \notin q$, *then* $[\Gamma \mid \Sigma]^{\varphi \setminus \ell} \vdash v : T^{q}$.

This lemma justifies removing a scoped location from φ at scope exit. Once t_2 reduces to a value v, the premise $\ell \notin q$ required by the typing rule (T-LOCIN) allows ℓ to be safely eliminated from the observation filter without affecting the typing of v.

5.3.5 Local Locations in Dynamic Typing. The final challenge is to ensure that scoped allocation and deallocation within one subterm do not affect unrelated subterms, up to typing. Consider an application concerning scopes, (ref v as x in t_2) t, where v is already a value. Reducing the left subterm introduces a local location ℓ in its elimination form, local ℓ in t_2. If both subterms were typed under the same observation filter, ℓ would incorrectly propagate to the typing of t, requiring general strengthening when ℓ is reclaimed before the reduction of t, which is impossible.

Dynamic Term Typing $\boxed{[\Gamma \mid \Sigma]^{\varphi} \vdash t : Q}$

$$\frac{c \in B}{[\Gamma \mid \Sigma]^{\varphi} \vdash c : B^{\varnothing}}\ (\textsc{t-cst}) \qquad \frac{x : T^{q} \in \Gamma \qquad x \in \varphi}{[\Gamma \mid \Sigma]^{\varphi} \vdash x : T^{x}}\ (\textsc{t-var})$$

$$\frac{[\Gamma \mid \Sigma]^{\varphi} \vdash t : T^{q} \qquad \blacklozenge \notin q}{[\Gamma \mid \Sigma]^{\varphi} \vdash \mathsf{ref}\ t : (\mathsf{Ref}\ [T^{q}])^{\blacklozenge}}\ (\textsc{t-ref}) \qquad \frac{[\Gamma \mid \Sigma]^{\varphi \ominus t_2} \vdash t_1 : T^{q} \qquad \blacklozenge \notin q \qquad [\Gamma \mid \Sigma]^{\varphi} \vdash t_2 : \mathsf{Ref}\ [U]^{p} \qquad \mathrm{LC}(t_1) \cap p = \varnothing}{[\Gamma \mid \Sigma]^{\varphi} \vdash \mathsf{ref}\ t_1\ \mathsf{at}\ t_2 : (\mathsf{Ref}\ [T^{q}])^{p}}\ (\textsc{t-refat})$$

$$\frac{[\Gamma \mid \Sigma]^{\varphi} \vdash t : \mathsf{Ref}\ [T^{p}]^{q} \qquad \blacklozenge \notin p \qquad p \subseteq \varphi \qquad \mathrm{LC}(t) \cap p = \varnothing}{[\Gamma \mid \Sigma]^{\varphi} \vdash\ !t : T^{p}}\ (\textsc{t-deref}) \qquad \frac{[\Gamma \mid \Sigma]^{\varphi} \vdash t_1 : \mathsf{Ref}\ [T^{p}]^{q} \qquad \blacklozenge \notin p \qquad [\Gamma \mid \Sigma]^{\varphi \ominus t_1} \vdash t_2 : T^{p} \qquad \mathrm{LC}(t_2) \cap q = \varnothing}{[\Gamma \mid \Sigma]^{\varphi} \vdash t_1 := t_2 : \mathsf{Unit}^{\varnothing}}\ (\textsc{t-assgn})$$

$$\frac{[\Gamma,\ f : F,\ x : P \mid \Sigma]^{q,x,f} \vdash t : Q \qquad F = (f(x : P) \to Q)^{q} \qquad q \subseteq \varphi \qquad \mathrm{LC}(t) \cap q = \varnothing}{[\Gamma \mid \Sigma]^{\varphi} \vdash \lambda f(x).t : F}\ (\textsc{t-abs})$$

$$\frac{[\Gamma,\ f : F,\ X^{x} <: P \mid \Sigma]^{q,x,f} \vdash t : Q \qquad F = (\forall f(X^{x} <: P).Q)^{q} \qquad q \subseteq \varphi \qquad \mathrm{LC}(t) \cap q = \varnothing}{[\Gamma \mid \Sigma]^{\varphi} \vdash \Lambda f(X^{x}).t : F}\ (\textsc{t-tabs})$$

$$\frac{[\Gamma \mid \Sigma]^{\varphi} \vdash t_1 : (f(x : T^{p}) \to Q)^{q} \qquad Q = U^{r} \qquad [\Gamma \mid \Sigma]^{\varphi \ominus t_1} \vdash t_2 : T^{p} \qquad \blacklozenge \notin p \qquad f \notin \mathrm{fv}(U) \qquad r \subseteq \blacklozenge\varphi, x, f \qquad \mathrm{LC}(t_2) \cap q, r = \varnothing \qquad \mathrm{LC}(t_1) \cap r = \varnothing}{[\Gamma \mid \Sigma]^{\varphi} \vdash t_1\ t_2 : Q[p/x, q/f]}\ (\textsc{t-app})$$

$$\frac{[\Gamma \mid \Sigma]^{\varphi} \vdash t : (\forall f(X^{x} <: T^{p}).Q)^{q} \qquad Q = U^{r} \qquad p \subseteq \varphi \qquad \blacklozenge \notin p \qquad f \notin \mathrm{fv}(U) \qquad r \subseteq \blacklozenge\varphi, x, f \qquad \mathrm{LC}(t) \cap p, r = \varnothing}{[\Gamma \mid \Sigma]^{\varphi} \vdash t[T^{p}] : Q[T^{p}/X^{x}, q/f]}\ (\textsc{t-tapp})$$

$$\frac{[\Gamma \mid \Sigma]^{\varphi} \vdash t_1 : \left(f(x : T^{p \oplus q}) \to Q\right)^{q} \qquad Q = U^{r} \qquad [\Gamma \mid \Sigma]^{\varphi \ominus t_1} \vdash t_2 : T^{p} \qquad \blacklozenge \in p \Rightarrow x \notin \mathrm{fv}(U) \qquad r \subseteq \blacklozenge\varphi, x, f \qquad f \notin \mathrm{fv}(U) \qquad \mathrm{LC}(t_2) \cap q, r = \varnothing \qquad \mathrm{LC}(t_1) \cap (p \oplus q), r = \varnothing}{[\Gamma \mid \Sigma]^{\varphi} \vdash t_1\ t_2 : Q[p/x, q/f]}\ (\textsc{t-app}^{\blacklozenge})$$

$$\frac{[\Gamma \mid \Sigma]^{\varphi} \vdash t : \left(\forall f(X^{x} <: T^{p \oplus q}).Q\right)^{q} \qquad Q = U^{r} \qquad p \subseteq \varphi \qquad \blacklozenge \in p \Rightarrow x \notin \mathrm{fv}(U) \qquad r \subseteq \blacklozenge\varphi, x, f \qquad f \notin \mathrm{fv}(U) \qquad \mathrm{LC}(t) \cap (p \oplus q), p, r = \varnothing}{[\Gamma \mid \Sigma]^{\varphi} \vdash t[T^{p}] : Q[T^{p}/X^{x}, q/f]}\ (\textsc{t-tapp}^{\blacklozenge})$$

$$\frac{[\Gamma \mid \Sigma]^{\varphi} \vdash t : Q \qquad \Gamma \vdash Q <: T^{q} \qquad q \subseteq \blacklozenge\varphi \qquad \mathrm{LC}(t) \cap q = \varnothing}{[\Gamma \mid \Sigma]^{\varphi} \vdash t : T^{q}}\ (\textsc{t-sub}) \qquad \frac{[\Gamma \mid \Sigma]^{\varphi} \vdash t_1 : T^{p} \qquad \blacklozenge \notin p \qquad x \notin \mathrm{fv}(Q) \qquad [\Gamma,\ x : \mathsf{Ref}\ [T^{p}]^{\blacklozenge} \mid \Sigma]^{\varphi \ominus t_1, x} \vdash t_2 : Q}{[\Gamma \mid \Sigma]^{\varphi} \vdash \mathsf{ref}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : Q}\ (\textsc{t-refin})$$

$$\frac{\Sigma(\ell, o) = T^{q} \qquad \ell \in \varphi \qquad \blacklozenge \notin q}{[\Gamma \mid \Sigma]^{\varphi} \vdash (\ell, o) : \mathsf{Ref}\ [T^{q}]^{\ell}}\ (\textsc{t-loc}) \qquad \frac{[\Gamma \mid \Sigma]^{\varphi} \vdash t : T^{q} \qquad \ell \notin q}{[\Gamma \mid \Sigma]^{\varphi} \vdash \mathsf{local}\ \ell\ \mathsf{in}\ t : T^{q}}\ (\textsc{t-locin})$$

Fig. 12. Term typing for program dynamics of $\{\mathsf{A}\}^{\blacklozenge}_{<:}$ with store typing. The content marked in violet are hidden and implicitly inferred for a static program with an empty store.

To address this, dynamic typing (Figure 12) distinguishes *local locations* (Figure 14)—locations that will be reclaimed in the future. For example in rule (T-APP), typing of t_2 explicitly excludes the local locations of t_1 in its observation filter, which is sound because those locations are guaranteed to be reclaimed before t_2 reduces. Conversely, local locations of t_2 do not affect typing of t_1, since t_1 is already a value, according to the reduction order described by *well-stepped terms* (Figure 14).

Together, these mechanisms establish deallocation safety: scoped references and their store locations are safely removed from observation filters at scope exit and are never reintroduced during subsequent reduction, ensuring they remain unreachable downstream.

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

$$\frac{c \in B}{\Gamma^{\varphi} \vdash c : B^{\varnothing}} \ (\textsc{ts-cst}) \qquad \frac{x : T^{q} \in \Gamma \qquad x \in \varphi}{\Gamma^{\varphi} \vdash x : T^{x}} \ (\textsc{ts-var})$$

$$\frac{\Gamma^{\varphi} \vdash t : T^{q} \qquad \blacklozenge \notin q}{\Gamma^{\varphi} \vdash \mathsf{ref}\ t : (\mathsf{Ref}\ [T^{q}])^{\blacklozenge}} \ (\textsc{ts-ref}) \qquad \frac{\Gamma^{\varphi} \vdash t_1 : T^{q} \qquad \blacklozenge \notin p, q \quad \Gamma^{\varphi} \vdash t_2 : \mathsf{Ref}\ [U]^{p}}{\Gamma^{\varphi} \vdash \mathsf{ref}\ t_1\ \mathsf{at}\ t_2 : (\mathsf{Ref}\ [T^{q}])^{p}} \ (\textsc{ts-refat})$$

$$\frac{\Gamma^{\varphi} \vdash t : \mathsf{Ref}\ [T^{p}]^{q} \quad \blacklozenge \notin p \qquad p \subseteq \varphi}{\Gamma^{\varphi} \vdash\ !t : T^{p}} \ (\textsc{ts-deref}) \qquad \frac{\Gamma^{\varphi} \vdash t_1 : \mathsf{Ref}\ [T^{p}]^{q} \quad \Gamma^{\varphi} \vdash t_2 : T^{p} \qquad \blacklozenge \notin p}{\Gamma^{\varphi} \vdash t_1 \coloneqq t_2 : \mathsf{Unit}^{\varnothing}} \ (\textsc{ts-assgn})$$

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

$$\frac{\Gamma^{\varphi} \vdash t_1 : (f(x : T^{p}) \to Q)^{q} \quad Q = U^{r} \quad \Gamma^{\varphi} \vdash t_2 : T^{p} \quad \blacklozenge \notin p \quad f \notin \mathrm{fv}(U) \quad r \subseteq \blacklozenge\varphi, x, f}{\Gamma^{\varphi} \vdash t_1\ t_2 : Q[p/x, q/f]} \ (\textsc{ts-app}) \qquad \frac{\Gamma^{\varphi} \vdash t : (\forall f(X^{x} <: T^{p}).Q)^{q} \quad Q = U^{r} \quad p \subseteq \varphi \quad \blacklozenge \notin p \quad f \notin \mathrm{fv}(U) \quad r \subseteq \blacklozenge\varphi, x, f}{\Gamma^{\varphi} \vdash t[T^{p}] : Q[T^{p}/X^{x}, q/f]} \ (\textsc{ts-tapp})$$

$$\frac{\Gamma^{\varphi} \vdash t_1 : \left(f(x : T^{p \curlyvee q}) \to Q\right)^{q} \quad \Gamma^{\varphi} \vdash t_2 : T^{p} \quad \blacklozenge \in p \Rightarrow x \notin \mathrm{fv}(U) \quad Q = U^{r} \quad r \subseteq \blacklozenge\varphi, x, f \quad f \notin \mathrm{fv}(U)}{\Gamma^{\varphi} \vdash t_1\ t_2 : Q[p/x, q/f]} \ (\textsc{ts-app}^{\blacklozenge}) \qquad \frac{\Gamma^{\varphi} \vdash t : \left(\forall f(X^{x} <: T^{p \curlyvee q}).Q\right)^{q} \quad p \subseteq \varphi \quad \blacklozenge \in p \Rightarrow x \notin \mathrm{fv}(U) \quad Q = U^{r} \quad r \subseteq \blacklozenge\varphi, x, f \quad f \notin \mathrm{fv}(U)}{\Gamma^{\varphi} \vdash t[T^{p}] : Q[T^{p}/X^{x}, q/f]} \ (\textsc{ts-tapp}^{\blacklozenge})$$

$$\frac{\Gamma^{\varphi} \vdash t : Q \quad \Gamma \vdash Q <: T^{q} \qquad q \subseteq \blacklozenge\varphi}{\Gamma^{\varphi} \vdash t : T^{q}} \ (\textsc{ts-sub}) \qquad \frac{\Gamma^{\varphi} \vdash t_1 : T^{p} \quad \blacklozenge \notin p \quad x \notin \mathrm{fv}(Q) \quad (\Gamma\ ,\ x : \mathsf{Ref}\ [T^{p}]^{\blacklozenge})^{\varphi,x} \vdash t_2 : Q}{\Gamma^{\varphi} \vdash \mathsf{ref}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : Q} \ (\textsc{ts-refin})$$

Fig. 13. Static term typing for programs in $\{\mathsf{A}\}^{\blacklozenge}_{<:}$ with an empty store.

5.4 Formal Definitions and Invariants

After Section 5.3 presents the intuition behind deallocation reasoning, we now formalize the definitions and invariants.

Figure 13 presents the static typing. Figure 14 defines the computation of *local locations* and the notion of *well-stepped terms* (Figure 14), which constrain how local locations may appear during reduction.

Dynamic Typing. Figure 12 defines the dynamic typing rules of $\{\mathsf{A}\}^{\blacklozenge}_{<:}$-calculus, formalizing the isolation principle described above: local locations of one subterm must not influence the observations or qualifiers of unrelated subterms. Concretely, each typing rule subtracts the local locations of sibling subterms from the observation filter.

For example, rule (T-APP) types t_2 under φ minus the local locations of t_1, so that allocations and deallocations within t_1 do not affect the typing of t_2. Once t_1 is fully reduced to a value, its local location set is empty by the constraints on *well-stepped terms*, and thus t_2 is typed with the original φ. Since local locations are always excluded from the observation filter, deallocated locations are never reintroduced.

Environment Update. Figure 15 formalizes environment evolution under reduction, refining the well-formedness of $\mathsf{A}^{\blacklozenge}_{<:}$-calculus. We introduce a meta-level variable κ recording reclaimed locations,

Local Locations and Qualifier Operations $\boxed{\mathsf{LC}(t) \in \mathcal{P}_{\mathsf{fin}}(\mathsf{Loc})}$

$$\mathsf{LC}(\mathsf{local}\ \ell\ \mathsf{in}\ t) := \ell, \mathsf{LC}(t) \qquad \mathsf{LC}(c),\ \mathsf{LC}(x),\ \mathsf{LC}((\ell, o)) := \varnothing$$
$$\mathsf{LC}(\lambda f(x).t),\ \mathsf{LC}(\mathsf{ref}\ t),\ \mathsf{LC}(!\ t),\ \mathsf{LC}(\Lambda f(X^{\chi}).t),\ \mathsf{LC}(t\ [Q]) := \mathsf{LC}(t)$$
$$\mathsf{LC}(t_1\ t_2),\ \mathsf{LC}(\mathsf{ref}\ t_1\ \mathsf{at}\ t_2),\ \mathsf{LC}(t_1 \coloneqq t_2),\ \mathsf{LC}(\mathsf{ref}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2) := \mathsf{LC}(t_1), \mathsf{LC}(t_2)$$
$$q \ominus t := q \setminus \mathsf{LC}(t)$$

Well-Stepped Terms $\boxed{t\ \mathsf{wf}}$

$$\frac{t\ \mathsf{wf}}{\mathsf{local}\ \ell\ \mathsf{in}\ t\ \mathsf{wf}} \qquad \frac{t_1\ \mathsf{wf} \quad t_2\ \mathsf{wf} \quad \mathsf{LC}(t_2) = \varnothing}{\mathsf{ref}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2\ \mathsf{wf}} \qquad \frac{t_1\ \mathsf{wf} \quad t_2\ \mathsf{wf} \quad \mathsf{value}\ t_1 \vee \mathsf{LC}(t_2) = \varnothing}{t_1\ t_2\ \mathsf{wf}}$$

$$\frac{t\ \mathsf{wf}}{\mathsf{ref}\ t\ \mathsf{wf}} \qquad \frac{t_1\ \mathsf{wf} \quad t_2\ \mathsf{wf} \quad \mathsf{LC}(t_1) = \varnothing \vee \mathsf{value}\ t_2}{\mathsf{ref}\ t_1\ \mathsf{at}\ t_2\ \mathsf{wf}} \qquad \frac{t_1\ \mathsf{wf} \quad t_2\ \mathsf{wf} \quad \mathsf{value}\ t_1 \vee \mathsf{LC}(t_2) = \varnothing}{t_1 \coloneqq t_2\ \mathsf{wf}}$$

$$\frac{\mathsf{LC}(t) = \varnothing}{\lambda f(x).t\ \mathsf{wf}} \qquad \frac{t\ \mathsf{wf}}{!\ t\ \mathsf{wf}} \qquad \frac{}{\mathsf{unit}\ \mathsf{wf}} \qquad \frac{}{(\ell, o)\ \mathsf{wf}} \qquad \frac{}{x\ \mathsf{wf}} \qquad \frac{t\ \mathsf{wf}}{\Lambda f(X^{\chi}).t\ \mathsf{wf}} \qquad \frac{t\ \mathsf{wf}}{t\ [Q]\ \mathsf{wf}}$$

Fig. 14. The local location computation and well-stepped terms in $\{\mathsf{A}\}^{\blacklozenge}_{<:}$-calculus.

Environment Update $\boxed{\kappa}\ \boxed{\Sigma, \varphi, \kappa \rightsquigarrow_t \Sigma', \varphi', \kappa'}$

$$\kappa \in \mathcal{P}_{\mathsf{fin}}(\mathsf{Loc}) \qquad \text{Already-Killed Locations}$$

(EU-BASE) $$\frac{\mathsf{dom}\ell(\Sigma) = \mathsf{dom}\ell(\Sigma')}{\Sigma, \varphi, \kappa \rightsquigarrow_t \Sigma', \varphi, \kappa}$$

(EU-FRESH) $$\frac{\kappa \cap q = \varnothing}{\Sigma, \varphi, \kappa \rightsquigarrow_t (\Sigma,\ (\ell, 0) : T^{q}), (\varphi,\ \ell), \kappa}$$

(EU-KILL) $$\frac{\ell \in \mathsf{dom}\ell(\Sigma) \quad \ell \in \mathsf{LC}(t)}{\Sigma, \varphi, \kappa \rightsquigarrow_t \Sigma, (\varphi \setminus \ell), (\kappa,\ \ell)}$$

Well-Formed and Well-Typed Stores $\boxed{\Sigma\ \mathsf{ok}}\ \boxed{[\Gamma \mid \Sigma]^{\varphi} \vdash \sigma \mid \kappa}$

$$[\Gamma \mid \Sigma]^{\varphi} \vdash \sigma \mid \kappa := \varphi \subseteq \mathsf{dom}\ell(\sigma) \subseteq \mathsf{dom}\ell(\Sigma) \wedge \varphi \cap \kappa = \varnothing \wedge$$
$$\forall \ell \in \kappa,\ \forall o,\ (\ell, o) \in \mathsf{dom}(\sigma),\ \sigma(\ell, o) = \mathsf{None} \wedge$$
$$\forall \ell \in \varphi,\ \forall o,\ (\ell, o) \in \mathsf{dom}(\sigma),\ \exists v,\ \sigma(\ell, o) = v \wedge v\ \mathsf{wf} \wedge$$
$$\forall T, q,\ \Sigma(\ell, o) = T^{q},\ q \subseteq \varphi \rightarrow [\Gamma \mid \Sigma]^{\varphi} \vdash \sigma(\ell, o) : \Sigma(\ell, o)$$

$$\frac{}{\varnothing\ \mathsf{ok}} \qquad \frac{\Sigma\ \mathsf{ok} \quad \mathsf{fv}(T) = \varnothing \quad \mathsf{ftv}(T) = \varnothing \quad q \in \mathsf{dom}\ell(\Sigma) \quad o > 0 \Rightarrow \ell \in \mathsf{dom}\ell(\Sigma)}{\Sigma,\ (\ell, o) : T^{q}\ \mathsf{ok}}$$

Fig. 15. The environment invariants of the $\{\mathsf{A}\}^{\blacklozenge}_{<:}$-calculus.

which is derivable from the store state. Unlike prior reachability type systems, which can arbitrarily choose larger context filters for preservation, we strictly specify how observation filters evolve with respect to the store and κ. We distinguish three cases:

(1) (EU-FRESH): Allocation of a fresh arena, either permanently or locally, grows the observation φ.

(2) (EU-KILL): Deallocation of a local arena shrinks φ and extends κ with the same location. Only local locations captured by the pre-reduction term can be freed.

(3) (EU-BASE): Other reductions that do not change the observation φ. Coallocation also falls into this category.

This is necessary because of the invariant for safe deallocation that the observation filter φ must remain disjoint from killed locations κ. Formalizing the environment update refines the proof structure for deriving more complicated results; however, it introduces notable extra engineering complexity compared with prior work, and we refer interested readers to our artifact for details.

Store Invariants. In Figure 15, the store invariants describe well-formed store typing, reachability tracking, and deallocation. The well-formed store Σ ok captures the *closedness* of each cell in the store typing: the type of each store object must be derivable from existing resources at the point it is appended to Σ. Because the store is two-dimensional, it does not naturally follow the telescoping structure as in prior reachability type systems [Deng et al. 2025a; Wei et al. 2024], resulting in a weaker and more general store invariant.

The well-typed store judgment, $[\Gamma \mid \Sigma]^{\varphi} \vdash \sigma \mid \kappa$, is the most critical invariant for type and store safety. For a store σ, the well-typed store requires: (1) the observation filter φ is closed and disjoint from κ; (2) locations in κ are reclaimed; and (3) objects having locations in φ are typeable with the types recorded in the store typing Σ. Crucially, the disjointedness between κ and φ guarantees that well-typed terms cannot observe deallocated resources. We only require store locations in φ to agree with the store typing Σ, as the remainder are either reclaimed or irrelevant to typing.

5.5 Metatheory

We establish syntactic type soundness for the $\{\mathrm{A}\}^{\blacklozenge}_{<:}$-calculus via progress and preservation theorems. Our proof structure follows prior work [Deng et al. 2025a; Wei et al. 2024], with lemma statements and definitions refined to account for deallocation reasoning.

5.5.1 Observability Properties. The observation is the key mechanism to reason about the resources necessary to type a term. Reasoning about the separation between observation and killed locations can implicitly guarantee the absence of killed locations in the term typing without exposing the κ. The following lemmas are proved by induction over the term typing.

Lemma 5.3 (Observability). *Term typing cannot assign qualifiers beyond observation,* i.e.*, if* $[\Gamma \mid \Sigma]^{\varphi} \vdash t : T^{q}$*, then* $q \subseteq \blacklozenge\varphi$.

Assigned qualifiers to well-typed values are lower bounds for observation in their typing.

Lemma 5.4 (Values are Non-Fresh). *If* $[\Gamma \mid \Sigma]^{\varphi} \vdash v : T^{q}$*, then* $[\Gamma \mid \Sigma]^{\varphi} \vdash v : T^{q \setminus \blacklozenge}$.

In the mechanized proof, we define an invertible value typing that inlines subtyping, eliminating the need for handling subsumption induction cases. These lemmas are then proved by induction on the invertible value typing.

5.5.2 Well-Stepped Terms. The well-stepped terms allow us to only reason about the terms reduced from some statically well-typed terms. The following lemmas are independent of the typing environment, so can be proved by induction over term reduction.

Lemma 5.5 (Well-Stepped Values). *If* v wf*, then* $\mathsf{LC}(t) = \varnothing$.

A well-stepped term always reduces to a well-stepped term under a well-formed store.

Lemma 5.6 (Well-Stepping of Terms). *If* t wf*, and* $[\varnothing \mid \Sigma]^{\varphi} \vdash \sigma \mid \kappa$ *for some* φ, Σ, κ*, and* $t \mid \sigma \longrightarrow t' \mid \sigma'$*, then* t' wf.

Syntactically correct but semantically ill-formed terms are excluded by the well-stepped term judgement, including the existence of scope eliminations in fully reduced values or not yet reduced terms.

5.5.3 Local Locations. The dynamic typing rules enforce the separation of the local locations of a term and its qualifier. The lemma below is proved by induction over dynamic typing.

Lemma 5.7 (Non-Escaping of Local Locations). *If* $[\Gamma \mid \Sigma]^{\varphi} \vdash t : T^{q}$, *then* $\mathsf{LC}(t) \cap q = \varnothing$.

The combination of Theorem 5.7, Theorem 5.2, and Theorem 5.5 reasons about the safe kill. A well-stepped term with local locations reduces to a value with empty local locations, and can be retyped in a shrinking observation separate from its previous local locations before reduction.

5.5.4 Substitution. In the call-by-value reduction, substitutions required by β-reduction are only considered for values containing no term variables, called top-level substitutions. The substitution lemmas generally follow the proof tree of Deng et al. [2025a].

Lemma 5.8 (Substitution Preserves Transitive Closures). *If* $x : T^{q} \in \Gamma$, *and* $p, q \subseteq \blacklozenge\,\mathrm{dom}\ell(\Sigma)$, *and* $p \cap \blacklozenge\varphi \subseteq q$, *and* $r{*}\Gamma \subseteq \blacklozenge\varphi$, *then* $r\theta{*}\Gamma\theta \subseteq r{*}\Gamma\theta$, *with substitution* $\theta = [p/x]$.

Proof. By transitively applying subtyping, commutativity between transitive closures and substitutions, and monotonicity of qualifier substitution. In each iteration of transitive lookup on a variable, the qualifier substitution afterwards produces a qualifier no smaller than the substitution beforehand. □

Lemma 5.9 (Top-Level Substitution). *If* $[x : T^{q}, \Gamma \mid \Sigma]^{\varphi} \vdash t : Q$, *and* $[\varnothing \mid \Sigma]^{p} \vdash v : T^{p}$, *and* $p \cap \blacklozenge\varphi$, *and* $q = p \vee q = \blacklozenge p \cap r$, *additionally* $\mathsf{LC}(t) \cap p = \varnothing$, *then for the substitution* $\theta = [p/x]$, $[\Gamma\theta \mid \Sigma]^{\varphi\theta} \vdash t[v/x] : Q\theta$.

Proof. By induction over the term typing of t.

Specially in the case (T-APP$^{\blacklozenge}$) and (T-TAPP$^{\blacklozenge}$), applying the induction hypothesis requires $(p \diamondsuit q)\theta \subseteq p\theta \diamondsuit q\theta$ established from Theorem 5.8. In case (T-SUB), an analogous substitution lemma for subtyping and qualifier subtyping is required.

The condition $\mathsf{LC}(t) \cap p = \varnothing$ preserves that substitution cannot cause local locations escaping from their defined scope. □

5.5.5 Main Soundness Result.

Theorem 5.10 (Progress). *If* $[\varnothing \mid \Sigma]^{\varphi} \vdash t : Q$, *and* Σ ok, *and* t wf, *then either* t *is a value, or for any store* σ *and killed locations* κ *where* $[\varnothing \mid \Sigma]^{\varphi} \vdash \sigma \mid \kappa$, *there exists a term* t' *and store* σ' *such that* $t \mid \sigma \longrightarrow t' \mid \sigma'$.

Proof. By induction over the term typing $[\varnothing \mid \Sigma]^{\varphi} \vdash t : Q$. □

Theorem 5.11 (Preservation). *If* $[\varnothing \mid \Sigma]^{\varphi} \vdash t : T^{q}$, *and* Σ ok, *and* t wf, *and* $t \mid \sigma \longrightarrow t' \mid \sigma'$ *for some* t' σ', *and* $[\varnothing \mid \Sigma]^{\varphi} \vdash \sigma \mid \kappa$ *for some* κ, *then there exists* Σ' φ' κ' *such that* $\Sigma, \varphi, \kappa \rightsquigarrow_{t} \Sigma', \varphi', \kappa'$, *and* $[\varnothing \mid \Sigma']^{\varphi'} \vdash \sigma' \mid \kappa'$, *and* $[\varnothing \mid \Sigma']^{\varphi'} \vdash t' : T^{q[p/\blacklozenge]}$ *for* $p \subseteq \mathrm{dom}\ell(\Sigma' \setminus \Sigma)$.

Proof. By induction over the typing $[\varnothing \mid \Sigma]^{\varphi} \vdash t : T^{q}$ and subsequently induction over the environment update $\Sigma, \varphi, \kappa \rightsquigarrow_{t} \Sigma', \varphi', \kappa'$.

For case (T-APP) with two sub-terms, the observation $\varphi \ominus t_1$ is distinguished separately for reduction of t_1 or t_2:

1. In the left congruence case of reduction on t_1, t_2 has not been reduced yet. The observation $\varphi \ominus t_1$ grows monotonically with the permanent allocation during the reduction of t_1, but will not reflect the shrinking because all killed locations are already within the local locations of t_1.

2. In the right congruence case of reduction on t_2, t_1 has already been reduced to a value. By Theorem 5.5, t_1 has no local locations, so the observation of t_2 becomes φ. The growth of observation by reduction on t_2 does not reflect on the typing of t_1 by Theorem 5.1.

3. In the contraction case where t_1 and t_2 are both values, all the local locations have already been killed by Theorem 5.5. The substitution Theorem 5.9 is applied twice for argument x and self-reference f to establish the case.

Other cases with two sub-terms handle local locations similarly. The contraction case of scope elimination local ℓ in v completes the proof by applying Theorem 5.2 to shrink the observation. □

All results are mechanized in Rocq; we refer readers to our artifact for details.

6 Case Studies

In this section, we demonstrate expressiveness through three case studies: relaxation of telescoping structures enabling a general fixed-point combinator without cyclic references (Section 6.1); non-lexical arenas for resource management (Section 6.2); and safe construction and reclamation of cyclic store structures with multi-hop cycles (Section 6.3).

6.1 General Fixed-Point Combinator

A general fixed-point combinator is typically implemented by storing a function in a mutable reference that refers to itself. Deng et al. [2025a] model this with cyclic references: the referent can observe the reference itself, permitting self-cycles in the store topology.

Such cyclic references impose extra constraints. In particular, the reference's outer qualifier must remain fixed to avoid enlarging the referent's reachability, which would violate telescoping well-formedness. Consequently, Deng et al. [2025a] restrict deep substitution[3] of function arguments to empty or singleton qualifiers, limiting expressiveness.

By contrast, our growable arenas with coarse-grained reachability permit recursive patterns without cyclic references, avoiding this limitation:

```
val a = new Ref()
def fix[T] (f: (g: (T -> T)^a -> (T -> T)^g)) : (T -> T)^a = {
  val c = new Ref(x => x) at a        // : Ref[(T -> T)^a]^c
  c := f((n : T) => (!c)(n))          // argument : (T -> T)^c <: (T -> T)^a
  !c                                  // : (T -> T)^a
}
```

Instead of allocating a fresh reference per recursive call, the function is stored in `c`, which is coallocated within arena `a`. Since all objects in `a` share a common reachability identity, internal self-references such as `c` are permitted without violating telescoping constraints. We implement the fixed-point operator and a factorial example in the $A^{\blacklozenge}_{<:}$-calculus (with a trivial integer extension); we refer readers to our artifact for details.

6.2 Callback Registration

In event-driven systems, callbacks enable asynchronous programming and flexible control flow. While typically registered during initialization, callbacks may be invoked long after registration and outside their defining scope. Delayed invocation requires callback resources to remain alive until explicit deregistration. To achieve this, callbacks are stored in a non-lexical resource pool that outlives their defining scope and is encapsulated by the registration closure to prevent misuse.

Non-lexical arenas with coarse-grained reachability satisfy these requirements. Handlers are coallocated in a single arena captured by the registration closure, which remains opaque to users. We sketch the callback registration API as follows:

```
val makeHandler = {
```

[3]See discussion in Section 6, Deng et al. [2025a]. Compared with rules (T-APP) and (T-APP♦) in Wei et al. [2024].

```
  val rp = new Ref(())                  // non-lexical resource pool
  (cb: Int => Unit) => {
    val h = new Ref(cb) at rp           // : Ref[(Int => Unit)]^h
    h                                   // return handler
  }                                     // : ((cb: Int => Unit) => Ref[Int => Unit]^rp)^rp
}
val f = (x: Int) => ()                  // : (Int => Unit)^f
makeHandler(f)                          // : Ref[Int => Unit]^makeHandler
```

The arena `rp` is a shared pool for registered callbacks. Each call to `makeHandler` returns a handler coallocated in `rp` and encapsulated by the closure. Crucially, callbacks are tracked via reachability to `makeHandler`, and deallocating `makeHandler` reclaims all handlers in bulk.

6.3 Circular Store Structures

Reference counting is a popular memory management technique in which each object maintains a count of active references pointing to it. When the count drops to zero, the object is automatically reclaimed. Though simple and efficient in many cases, reference counting fails to reclaim objects forming multi-hop cycles where none of the reference counts ever reaches zero.

Our system properly handles both the construction and safe deallocation of such cyclic structures:

```
{ // scope starts
  val a = new Ref(()) scoped            // : Ref[Unit]^a
  val f = (x : Int) => 7                // : (Int => Int)^f <: (Int => Int)^a
  val c1, c2, c3 = new Ref(f) at a      // : Ref[(Int => Int)^a]^··
  c1 := x => { (!c2)(x) }               // OK, Ref[(Int => Int)^a]^c2 <: Ref[(Int => Int)^a]^a
  c2 := x => { (!c3)(x) }               // OK
  c3 := x => { (!c1)(x) }               // OK
} // deallocate {a, r1, r2, r3}
```

The references `c1`, `c2`, and `c3` form a transitive multi-hop cycle through the store. Because reachability is tracked at the arena level, the three references (`c1`, `c2`, and `c3`) coallocated with the proxy reference (`a`) are grouped in a single reachability unit. The entire structure behaves as a circular linked list and is treated as a whole under coarse-grained reachability tracking. At scope end, the scoped reference `a` and its coallocated references are deallocated in bulk, avoiding the pitfalls of reference counting while preserving timely and safe reclamation.

Circular patterns, where a reference must transitively reach previously allocated intermediaries, break telescoping structures and lie beyond prior reachability type systems [Deng et al. 2025a; Wei et al. 2024]. Our work addresses this challenge. Moreover, this design can generalize to doubly linked lists, which can be implemented via Church encoded pairs [Wei et al. 2024] storing two closures within each node.

7 Discussion and Limitations

Imprecision from Flow-Insensitivity. Our approach is type-based, meaning that reachability types and our deallocation reasoning atop are flow-insensitive, which in general strikes a good balance between precision and complexity like most type systems. Flow-insensitivity in our design arises from two sources: (1) reachability qualifiers themselves, which over-approximate reachable resources; and (2) deallocation reasoning based on the local locations over reduction (Section 5.3), which does not introduce extra imprecision.

Although our deallocation reasoning is lightweight in annotation, it is less expressive than flow-sensitive effect systems as informally described in Wei et al. [2024] and more generally by Gordon [2021]. Specifically, deallocation is guaranteed only for scoped arenas and at scope exit,

not for arbitrary arenas or in the middle of a scope. We regard flow-sensitive studies as parallel work, including recent representative efforts by Deng et al. [2025b] and Jia et al. [2025a].

Garbage Collection. Our system focuses on impure high-level languages where the choice of using garbage collection (GC) is common. We do not aim to remove GC entirely; instead, we provide the additional benefits of timely guaranteed reclamation via selective stack discipline when desired. This mechanism can ultimately apply to resources beyond reference cells, such as files or sockets. Our language design is also open to having internal GC within an arena as a complementary option.

Completely eliminating GC would be possible but too restrictive because it forbids all non-lexical arenas and escaping patterns. This is consistent with Rust's use of reference counting: although Rust's ownership system does not rely on GC, adding reference counting allows additional expressiveness such as sharing.

Type Inference and Empirical Implementation. Sound, expressive, and efficient type checking and inference remain open challenges for reachability types. In particular, typing an escaped term (Section 2.1, Section 3.2.3) is subject to the avoidance problem [Jia et al. 2025b]: a variable mentioned in types becomes ill-scoped when its defining scope ends and thus needs replacement. In this work and prior reachability type systems, such replacement requires explicit term-level coercions (omitted for clarity), rather than a more natural way to upcast via subtyping. Improving the subtyping and deriving typing algorithms are important but orthogonal to this work.

Data Structures. In this work, we have shown how to encode cyclic store structures with multi-hop cycles, a key step towards supporting practical cyclic data structures. Similar goals have appeared in related works: Xu et al. [2024] demonstrates that capturing types can model mutable, cyclic object graphs via class extensions, though they stop at providing the full details of the class extensions. We likewise leave a complete object encoding to future work.

8 Related Work

Regions/Arenas. Early forms of regions/arenas were explored under various names with explicit management, but without static safety guarantees. Hanson [1990] introduced *arenas* for fast allocation and deallocation by manually grouping objects by lifetime. Utting [1997] explored a similar idea for ease of reasoning, called *local stores*. Gay and Aiken [1998] identified safety limitations in these systems and introduced a mechanism that skips deallocation for any region still referenced. Their runtime approach demonstrated performance advantages, especially for list structures, despite incurring reference counting overhead. A follow-up study [Gay and Aiken 2001] explored annotating structural information to mitigate the cost of reference counting.

Prominent examples of region-based systems include ML Kit [Tofte et al. 2001; Tofte and Talpin 1997] and Cyclone [Grossman et al. 2002]. Both enforce static safety but differ in languages and styles: ML Kit targets on a functional language with *implicit* region management via a Hindley-Milner style inference, while Cyclone adopts *explicit* regions in a low-level C-like language augmented with existential types to encode closures. In this work, arenas require explicit allocation and scoping, but unlike traditional explicit systems, regions keys or handles are not needed—existing references suffice. This allows region polymorphism to be naturally expressed as function abstraction.

Beyond surface languages, MLKit [Tofte et al. 2001] and Cyclone [Grossman et al. 2002] share foundational elements. Both rely on forms of effect (liveness) types for static guarantees, and following the insight [Wadler and Thiemann 2003] that effects can be transposed as monads, both can be embedded in a monadic region calculus [Fluet and Morrisett 2004]. While lexically scoped lifetime have been noted as a limitation, both systems provide dynamically managed global regions to compensate, although this sacrifices some of the performance and isolation benefits of regions.

Further development in Cyclone [Fluet and Wang 2004; Hicks et al. 2004] introduced escaping arenas that require additional capability management. These extensions were later formalized and proven sound via linear regions [Fluet et al. 2006].

Region Logic. In assertion-based verification techniques, region logic [Banerjee et al. 2013] allows users to declare and update ghost fields and variables of type "region". Regions in their work store objects, and have been refined to storing memory locations [Bao et al. 2015, 2018]. In our work, the reachability qualifier of each arena serves a similar purpose: it provides an upper bound on the locations that may store values belonging to the arena. We envision that our approach could be leveraged to synthesize those statements (in region logic) through reachability qualifiers, thereby, reducing annotation overhead in their frameworks. We leave a proper investigation as future work.

Linear Types, Ownership Types, and Their Region Variants. Beyond regions, other mechanisms for static lifetime control exist in flow-sensitive forms. Linear type systems [Turner et al. 1995; Wadler 1990] require each variable to be used exactly once. While this prevents sharing, it streamlines reasoning about lifetime and deallocation. Ownership types [Clarke et al. 2013, 1998; Noble et al. 1998] are often designed in object-oriented languages, where variants differ in their topological restrictions and encapsulation disciplines. Rust [Klabnik and Nichols 2019] is a notable instance with growing adoption, enforcing unique ownership and a single mutable access path throughout execution. This model enables predictable memory deallocation. Meanwhile, such strict invariants can pose challenges for users [Crichton et al. 2023] and complicate the safe construction of cyclic data structures [Milano et al. 2022; Yanovski et al. 2021]. Proposals have been made for relaxing the uniqueness requirements in Rust [Noble et al. 2022] and ownership types [Müller and Rudich 2007].

Efforts to integrate regions with linear or ownership types aim to support both flexible sharing and non-lexical lifetime control. Linear regions [Fluet et al. 2006; Walker and Watkins 2001] track regions and resources in unrestricted ways, whereas region capabilities are tracked linearly, required for granting access and consumed by deallocation. Pony [Clebsch et al. 2017] employs implicit regions for actor programming, decomposing fractional capabilities for fine-grained control. Verona's region system, Reggio [Arvidsson et al. 2023], combines ownership with explicit regions to ensure isolation among threads in concurrent programming: each region can be managed by a configured strategy, and each thread operates within a window of a single mutable region. These systems generally require users to explicitly *enter* or *focus* on a region to obtain the necessary capabilities—a hallmark of flow-sensitive reasoning. Milano et al. [2022] formalized these mechanisms as *virtual transformations* and proposed an inference scheme to reduce the manual overhead.

Reachability Types. This work extends reachability types with new mechanisms for store management and scoping. Seeking to adapt separation logic [Reynolds 2002] to functional languages, Bao et al. [2021] introduced reachability qualifiers, sets of variables annotated on types, to express resource access patterns. Wei et al. [2024] refined this idea by incorporating *one-step* reachability and freshness, yielding a more precise and dynamic model. Their polymorphic calculus $\mathsf{F}^{\blacklozenge}_{<:}$ tracks resource sharing without enforcing global invariants, offering a flexible foundation for reasoning about resources in higher-order settings. Graph IR [Bracevac et al. 2023] used reachability types to optimize impure higher-order programs. We build upon this framework to develop our calculi.

The semantics of the monomorphic $\lambda^{\blacklozenge}$ were analyzed via logical relations [Bao et al. 2025], and, counterintuitively, proven terminating despite its extension with a higher-order store. This result stems from the store's adherence to the telescoping requirement (see Section 2.2), which further motivated efforts to enable cycles in the store [Deng et al. 2025a]. Their $\lambda_\circ$ allows self-cycles (see Figure 2b) and thus non-terminating computation (*e.g.*, fixed-point combinators), but imposes extra constraints on application rules to do so. By contrast, our $\mathsf{A}^{\blacklozenge}_{<:}$ allows similar patterns via intra-arena

cycles without such constraints, although cycles require at least two cells. Deng et al. [2025a] further proposed refinements to the reference model, which we incorporate in our design.

Reachability types are inherently flow-insensitive and thus cannot express linearity/uniqueness in isolation. Flow-sensitive effect extensions have been discussed regarding this limitation [Bao et al. 2021; Wei et al. 2024], including destructive effects for deallocation. These approaches would require additional effect qualifiers and corresponding reasoning in typing judgements. Our $\{A\}^{\blacklozenge}_{<:}$, in contrast, ensures scoped deallocation safety by parsing lifetime information directly from reachability qualifiers. To support explicit, flow-sensitive deallocation (*e.g.*, manually calling `free`), a separate flow-sensitive effect extension would still be required.

With a similar design to reachability types, capturing types [Boruch-Gruszecki et al. 2023; Xu et al. 2024] were initially proposed for effect safety in Scala [Odersky et al. 2021, 2022]. Their design allows to work around the telescoping requirement (see Section 2.2) by employing a top qualifier `{cap}`. Top qualifiers, however, similar to *top types*, offer little information in reasoning, as a resource tracked as `{cap}` can potentially reach/capture anything.

9 Conclusion

In this work, we unified reachability types, arena-based resource management, and stack discipline. We introduced the $A^{\blacklozenge}_{<:}$-calculus, which generalizes reachability reasoning to non-lexical shadow arenas over a two-dimensional store model. By lifting reachability tracking from fine-grained to coarse-grained, $A^{\blacklozenge}_{<:}$ relaxes the telescoping structures that limit cyclic store structures. Building upon $A^{\blacklozenge}_{<:}$-calculus, the novel $\{A\}^{\blacklozenge}_{<:}$-calculus reestablishes selective stack discipline through scoped allocation, enabling sound, bulk, and flow-insensitive deallocation reasoning. Both calculi were formalized and proven type sound and memory safe in Rocq. This unified framework provides a lightweight yet expressive foundation for safe resource management in higher-order languages, reconciling first-class arenas and scoped lifetime control within a single type system.

Data Availability Statement

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

Acknowledgments

We thank Haotian Deng for related contributions to reachability types. This work was supported in part by NSF award 2348334 and Augusta faculty startup package, as well as gifts from Meta, Google, Microsoft, and VMware.

References

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 (2023), 1363–1393. doi:10.1145/3622846

Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 2013. Local Reasoning for Global Invariants, Part I: Region Logic. *J. ACM* 60, 3 (2013), 18:1–18:56. doi:10.1145/2485982

Yuyan Bao, Songlin Jia, Guannan Wei, Oliver Bracevac, and Tiark Rompf. 2025. Modeling Reachability Types with Logical Relations: Semantic Type Soundness, Termination, Effect Safety, and Equational Theory. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 1837–1864. doi:10.1145/3763116

Yuyan Bao, Gary T. Leavens, and Gidon Ernst. 2015. Conditional effects in fine-grained region logic. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs, FTfJP 2015, Prague, Czech Republic, July 7, 2015*, Rosemary Monahan (Ed.). ACM, 5:1–5:6. doi:10.1145/2786536.2786537

Yuyan Bao, Gary T. Leavens, and Gidon Ernst. 2018. Unifying separation logic and region logic to allow interoperability. *Formal Aspects Comput.* 30, 3-4 (2018), 381–441. doi:10.1007/S00165-018-0455-5

Yuyan Bao, Guannan Wei, Oliver Bracevac, 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 (2021), 1–32. doi:10.1145/3485516

Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.* 45, 4 (2023), 21:1–21:52.

Oliver Bracevac, 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 (2023), 400–430. doi:10.1145/3622813

Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming*. Lecture Notes in Computer Science, Vol. 7850. Springer, 15–58.

David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1998, Vancouver, British Columbia, Canada, October 18-22, 1998*, Bjørn N. Freeman-Benson and Craig Chambers (Eds.). ACM, 48–64. doi:10.1145/286936.286947

Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. 2017. Orca: GC and type system co-design for actor languages. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 72:1–72:28. doi:10.1145/3133896

Will Crichton, Gavin Gray, and Shriram Krishnamurthi. 2023. A Grounded Conceptual Model for Ownership Types in Rust. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1224–1252.

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, Gilles Kahn (Ed.). ACM, 151–160. doi:10.1145/91556.91622

Haotian Deng, Siyuan He, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2025a. Complete the Cycle: Reachability Types with Expressive Cyclic References. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 3398–3425. doi:10.1145/3763172

Haotian Deng, Siyuan He, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2025b. Free to Move: Reachability Types with Flow-Sensitive Effects for Safe Deallocation and Ownership Transfer. arXiv:2510.08939 [cs.PL] https://arxiv.org/abs/2510.08939

Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. 2006. Linear Regions Are All You Need. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science)*, Peter Sestoft (Ed.). Springer, 7–21. doi:10.1007/11693024_2

Matthew Fluet and J. Gregory Morrisett. 2004. Monadic regions. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, Chris Okasaki and Kathleen Fisher (Eds.). ACM, 103–114. doi:10.1145/1016850.1016867

Matthew Fluet and Daniel Wang. 2004. Implementation and Performance Evaluation of a Safe Runtime System in Cyclone. In *SPACE*.

David Gay and Alex Aiken. 1998. Memory Management with Explicit Regions. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, Jack W. Davidson, Keith D. Cooper, and A. Michael Berman (Eds.). ACM, 313–323. doi:10.1145/277650.277748

David Gay and Alex Aiken. 2001. Language Support for Regions. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, Michael Burke and Mary Lou Soffa (Eds.). ACM, 70–80. doi:10.1145/378795.378815

Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021), 4:1–4:79. doi:10.1145/3450272

Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 282–293. doi:10.1145/512529.512563

D. R. Hanson. 1990. Fast allocation and deallocation of memory based on object lifetimes. *Softw. Pract. Exper.* 20, 1 (Jan. 1990), 5–12. doi:10.1002/spe.4380200104

Michael W. Hicks, J. Gregory Morrisett, Dan Grossman, and Trevor Jim. 2004. Experience with safe manual memory-management in cyclone. In *ISMM*. ACM, 73–84.

Songlin Jia, Craig Liu, Siyuan He, Haotian Deng, Yuyan Bao, and Tiark Rompf. 2025a. Typestate via Revocable Capabilities. arXiv:2510.08889 [cs.PL] https://arxiv.org/abs/2510.08889

Songlin Jia, Guannan Wei, Siyuan He, Yuyan Bao, and Tiark Rompf. 2025b. Escape with Your Self: Sound and Expressive Bidirectional Typing with Avoidance for Reachability Types. arXiv:2404.08217 [cs.PL] https://arxiv.org/abs/2404.08217

Steve Klabnik and Carol Nichols. 2019. *The Rust Programming Language*. No Starch Press.

Fengyun Liu, Sandro Stucki, Nada Amin, Paolo Giosuè Giarrusso, and Martin Odersky. 2020. Stoic: Towards Disciplined Capabilities. https://infoscience.epfl.ch/handle/20.500.14299/164482

John McCarthy. 1959. LISP: a programming system for symbolic manipulations. In *ACM National Meeting*. ACM, 1:1–1:4.

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

Peter Müller and Arsenii Rudich. 2007. Ownership transfer in universe types. In *OOPSLA*. ACM, 461–478.

James Noble, Julian Mackay, and Tobias Wrigstad. 2022. Rusty Links in Local Chains. In *Proceedings of the 24th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2022, Berlin, Germany, 7 June 2022*, Henrique Rebêlo (Ed.). ACM, 1–3. doi:10.1145/3611096.3611097

James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *ECOOP (Lecture Notes in Computer Science, Vol. 1445)*. Springer, 158–185.

Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondrej Lhoták. 2021. Safer exceptions for Scala. In *SCALA at SPLASH*. ACM, 1–11.

Martin Odersky, Aleksander Boruch-Gruszecki, Edward Lee, Jonathan Immanuel Brachthäuser, and Ondrej Lhoták. 2022. Scoped Capabilities for Polymorphic Effects. *CoRR* abs/2207.03402 (2022).

Leo Osvald, Grégory M. Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *OOPSLA*. ACM, 234–251.

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

Peter Thiemann. 2025. What I Always Wanted to Know about Second Class Values. In *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday* (Singapore, Singapore) *(OLIVIERFEST '25)*. Association for Computing Machinery, New York, NY, USA, 117–127. doi:10.1145/3759427.3760373

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, and Peter Sestoft. 2001. Programming with Regions in the ML Kit (for Version 4). (10 2001).

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based Memory Management. *Inf. Comput.* 132, 2 (1997), 109–176. doi:10.1006/INCO.1996.2613

David N. Turner, Philip Wadler, and Christian Mossin. 1995. Once Upon a Type. In *FPCA*. ACM, 1–11.

Mark Utting. 1997. Reasoning about aliasing. *Formal Aspects of Computing* 3 (1997), 1–15.

Philip Wadler. 1990. Linear Types can Change the World!. In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, Manfred Broy and Cliff B. Jones (Eds.). North-Holland, 561.

Philip Wadler and Peter Thiemann. 2003. The marriage of effects and monads. *ACM Trans. Comput. Log.* 4, 1 (2003), 1–32.

David Walker and Kevin Watkins. 2001. On Regions and Linear Types. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 181–192. doi:10.1145/507635.507658

Guannan Wei, Oliver Bracevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 393–424. doi:10.1145/3632856

Anxhelo Xhebraj, Oliver Bracevac, Guannan Wei, and Tiark Rompf. 2022. What If We Don't Pop the Stack? The Return of 2nd-Class Values. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, Berlin, Germany, June 6-10, 2022 (LIPIcs)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:29. doi:10.4230/LIPICS.ECOOP.2022.15

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 (2024), 1181–1207. doi:10.1145/3649853

Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: separating permissions from data in Rust. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30.