



 Latest updates: <https://dl.acm.org/doi/10.1145/3763172>

RESEARCH-ARTICLE

## Complete the Cycle: Reachability Types with Expressive Cyclic References

**HAOTIAN DENG**, Purdue University, West Lafayette, IN, United States

**SIYUAN HE**, Purdue University, West Lafayette, IN, United States

**SONGLIN JIA**, Purdue University, West Lafayette, IN, United States

**YUYAN BAO**, Augusta University, Augusta, GA, United States

**TIARK ROMPF**, Purdue University, West Lafayette, IN, United States

**Open Access Support** provided by:

**Purdue University**

**Augusta University**



PDF Download

3763172.pdf

31 March 2026

Total Citations: 1

Total Downloads: 175



Published: 09 October 2025

Accepted: 12 August 2025

Received: 26 March 2025

[Citation in BibTeX format](#)



# Complete the Cycle: Reachability Types with Expressive Cyclic References

HAOTIAN DENG, Purdue University, USA

SIYUAN HE, Purdue University, USA

SONGLIN JIA, Purdue University, USA

YUYAN BAO, Augusta University, USA

TIARK ROMPF, Purdue University, USA

Local reasoning about programs that combine aliasing and mutable state is a longstanding challenge. Existing approaches – ownership systems, linear and affine types, uniqueness types, and lexical effect tracking – impose global restrictions such as uniqueness or linearity, or rely on shallow syntactic analyses. These designs fall short with higher-order functions and shared mutable state. Reachability Types (RT) track aliasing and separation in higher-order programs, ensuring runtime safety and non-interference. However, RT systems face three key limitations: (1) they prohibit cyclic references, ruling out non-terminating computations and fixed-point combinators; (2) they require deep tracking, where a qualifier must include all transitively reachable locations, reducing precision and hindering optimizations like fine-grained parallelism; and (3) referent qualifier invariance prevents referents from escaping their allocation contexts, making reference factories inexpressible.

In this work, we address these limitations by extending RT with three mechanisms that enhance expressiveness. First, we introduce cyclic references, enabling recursive patterns to be encoded directly through the store. Second, we adopt shallow qualifier tracking, decoupling references from their transitively reachable values. Finally, we introduce an escaping rule with reference subtyping, allowing referent qualifiers to outlive their allocation context. These extensions are formalized in the  $F_{<}^{\circ}$ -calculus with a mechanized proof of type soundness, and case studies illustrate expressiveness through fixpoint combinators, non-interfering parallelism, and escaping read-only references.

CCS Concepts: • **Software and its engineering** → *Semantics*; **Functional languages**; **General programming languages**.

Additional Key Words and Phrases: reachability types, reference types, cyclic references, aliasing, type systems

## ACM Reference Format:

Haotian Deng, Siyuan He, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2025. Complete the Cycle: Reachability Types with Expressive Cyclic References. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 394 (October 2025), 28 pages. <https://doi.org/10.1145/3763172>

## 1 Introduction

Local reasoning with aliasing and mutable state is a foundational challenge in programming languages. This challenge inspired a range of ideas, from ownership systems [Clarke et al. 2013, 2001, 1998; Hogg 1991], to linear/affine types [Girard 1987; Wadler 1990], to uniqueness types [Barendsen and Smetsers 1996], to capability-based region type systems [Crary et al. 1999; Walker et al. 2000].

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



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/10-ART394

<https://doi.org/10.1145/3763172>

```

1 val c = ... // : Ref[(Unit => Unit)q]c
2 def f(x : Unit) = {(!c)(x)}
3 // + [ f : (Unit => Unit)c ]
4 c := f // c ✗: q
5 // Error! Referent qualifier mismatch

```

(a) Landin’s knot is not typeable in  $\lambda^\diamond$  [Wei et al. 2024] due to the lack of cyclic references. Line 4 fails to type check because the referent qualifier  $q$  can only reach observable resources before  $c$  is declared, which does not include  $c$ .

```

1 val c = ... // :  $\mu z$ .Ref[(Unit => Unit)z]c
2 def f(x : Unit) = {(!c)(x)}
3 // + [ f : (Unit => Unit)c ]
4 c := f
5 // Okay, because c is a cyclic reference

```

(b) Landin’s knot typed with a cyclic reference in  $F_{<}^o$  (this work). Line 4 is type checked with  $T$ -SASSGN-V (Figure 8). Its reachability graph is shown in Figure 6.

Fig. 1. Typing Cyclic References (Section 1.1): not possible in Wei et al. [2024] (Figure 1a), possible in this work (Figure 1b).

```

1 val inner = new Ref(...) // : Ref[...]inner
2 val outer = new Ref(inner)
3 // : Ref[Ref[...]inner]outer,inner
4 def par(b1: (Unit => Unit)♦
5         (b2: (Unit => Unit)♦) = ...
6 // parallelize b1 and b2
7 par { inner := ... } { outer := ... }
8 // Error! inner overlaps outer

```

(a)  $\lambda^\diamond$ ’s deep reference tracking always make reference track their referents. Line 7 fails to type check because  $inner$  is considered part of  $outer$ , making it impossible to parallelize uses of the two resources.

```

1 val inner = new Ref(...) // Ref[...]inner
2 val outer = new Ref(inner)
3 // : Ref[Ref[...]inner]outer
4 def par(b1: (Unit => Unit)♦
5         (b2: (Unit => Unit)♦) = ...
6 // parallelize b1 and b2
7 par { inner := ... } { outer := ... }
8 // Okay, inner and outer are disjoint

```

(b) Under shallow reference tracking in  $F_{<}^o$  (this work), line 7 is type checked because  $inner$  and  $outer$  are disjoint, thus can be safely parallelized.

Fig. 2. Qualifier separation between reference and referent (Section 1.2): not possible in Wei et al. [2024] (Figure 2a), possible in this work (Figure 2b).

These ideas have both shaped theory and influenced mainstream programming languages design. Notably, Rust [Matsakis and II 2014] has emerged as a gold standard for practical ownership-based systems, combining affine types and lexical lifetimes to ensure memory and concurrency safety. However, Rust enforces a strict aliasing discipline: its “shared XOR mutable” invariant requires global uniqueness of mutable references. This restricts expressiveness in idioms with shared mutable references, cyclic structures, or higher-order functions that close over mutable state.

Reachability Types (RT) [Bao et al. 2021; Wei et al. 2024] offer a promising alternative to bring expressive, fine-grained reasoning about resource sharing and separation to higher-level functional languages, inspired by separation logic [O’Hearn et al. 2001; Reynolds 2002]. Existing RT systems demonstrate partial success in reasoning about safety and separation across higher-order functions [Bao et al. 2021], mutable state, and polymorphic types [Wei et al. 2024]. For example, RT successfully type challenging closure patterns that share mutable state – beyond the reach of ownership, linearity, or lexical effect systems (See Figure 1 in Wei et al. [2024]).

However, the core of reference types in existing RT systems has received less attention and remains limited in several key aspects: (1) They lack support for cyclic references and recursive functions (Section 1.1); (2) their qualifiers overapproximate by tracking all transitively reachable heap locations (Section 1.2); (3) they disallow semantically valid reference escapes (Section 1.3).

## 1.1 Cyclic References

Recent theoretical advances in RT have shown that variants of RT that support higher-order references remain *terminating* [Bao et al. 2025]. This is surprising, because languages that combine

```

1 def mkRef() =
2   val x = ... // Tc
3   val c = new Ref(x) // Ref[Tx]c
4   c // Ref[Tx]c ✗: Ref[T∅]c
5 // Error! x is untracked outside mkRef

```

(a) Reference factory functions not typeable in  $\lambda^\diamond$ . Line 4 fails because the returning reference  $c$ 's referent contains local variable  $x$  which cannot escape the function scope.

```

1 def mkRef() =
2   val x = ... // Tc
3   val c = new Ref(x) // Ref[Tx]c
4   c // Okay, Ref[Tx]c <:  $\mu z$ .Ref[Tz]c
5 mkRef(...) // Okay,  $\mu z$ .Ref[Tz]\diamond

```

(b) A reference factory function typeable in  $F_{<}^\circ$ . (this work). Line 4 is successfully type checked by  $T$ -ESC, escaping to a read-only cyclic reference (Figure 13).

Fig. 3. Typing escaping references (Section 1.3): not possible in Wei et al. [2024] (Figure 3a), possible in this work (Figure 3b).

higher-order functions with higher-order mutable references typically permit general recursion by encoding fixed-point combinators through the store. The key pattern underlying such encodings is Landin's Knot [Landin 1964], where a function is stored in a reference that in turn is captured by the function itself, creating a cycle in the store, thereby enabling non-terminating computations.

A typical encoding of Landin's Knot is shown in Figure 1a. The program creates a mutable reference  $c$ , which initially stores a function value. The reference is updated with function  $f$  (line 4), which captures  $c$  and calls the function stored at reference  $c$ . Calling the function stored at  $c$  will recursively invoke the function itself through the store, leading to infinite recursion.

However, the code in Figure 1a is not typeable in [Wei et al. 2024]'s system, because in their system, referent qualifiers are invariant: a reference can only be updated with values having an *equivalent* qualifier. In this example, when  $c$  is allocated, its type must be assigned before  $c$  itself is in scope; and the initial qualifier  $q$  used in its type cannot include  $c$ , since  $c$  has not yet been bound. But the reference  $c$  is updated with a function that captures  $c$  (line 4), violating referent qualifier invariance. As a result, general recursion via Landin's Knot is disallowed in their system.

In this work, we introduce  $F_{<}^\circ$ , an extension of [Wei et al. 2024]'s  $F_{<}^\diamond$  system with explicit cyclic references. As shown in Figure 1b, variable  $c$  has a cyclic reference type: the  $\mu z$  binder may appear in the inner qualifier to refer back to the outer reference, making Landin's knot typeable.

Extending prior work [Wei et al. 2024] to support cyclic reference types introduces unique challenges: as we discuss in Section 2.2, a naive assignment rule for cyclic references is unsound. To preserve soundness, we impose two constraints on cyclic assignments: (1) the assigned reference term must be a variable, and (2) the assignee's qualifier must be a singleton matching that variable. A second challenge is adjusting the application rule so cyclic assignments in abstractions remain well-typed after  $\beta$ -reduction.

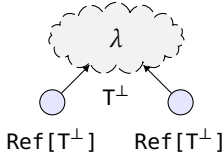
We describe our approach to overcoming these challenges in Section 2.2, and present a general fixpoint operator as case study in Section 5.1.

## 1.2 Flexible & Precise Reference Qualifiers

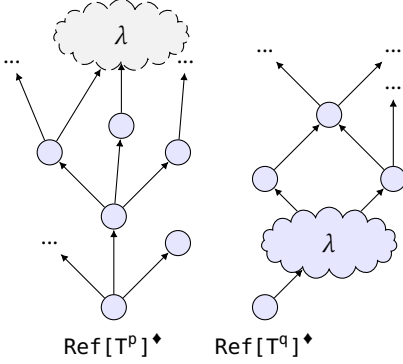
Wei et al. [2024]'s reference typing employs a “deep” tracking mechanism to enforce strong separation between portions of the heap. However, this limits the shape of heap structures that can be formed using nested references. As illustrated in Figure 4b, separate references<sup>1</sup> remain distinct from each other in terms of their *transitively reachable locations*.<sup>2</sup> This prevents parent references from sharing store locations, making heap structures with nested references less flexible.

<sup>1</sup>Separation is indicated by the freshness marker  $\diamond$ , which denotes contextual freshness. Resources with  $\diamond$  reach contextually fresh objects, making them separate from other observable resources. See Section 2.1 for details.

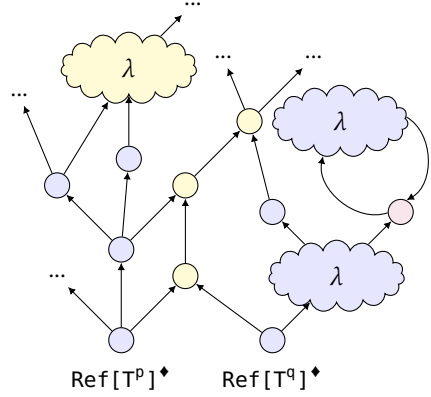
<sup>2</sup>This follows directly from  $T$ -REF, where  $q$  is included in the qualifiers of both the inner referent and the outer reference. As a result, the outer qualifier always includes all transitively reachable locations.



(a) Reachability graph of Bao et al. [2021]’s system. Because references are first-order with untracked referents (depicted as a dotted gray cloud), the system cannot represent hierarchical structures using reference types.



(b) Reachability graph illustrating the “deep separation” reference tracking model in Wei et al. [2024]’s system. References marked with ♦ are separate from each other and enforce strict separation of all transitively reachable locations. Untracked data, which separate references may still reach, is depicted as dotted gray clouds.



(c) Reachability graph illustrating the “shallow separation” reference tracking model in  $F_{<}^o$  (this work). Unlike Wei et al. [2024]’s system, our model permits separate references to share child nodes (depicted in yellow) and supports cyclic references where values can reach the references themselves (highlighted in purple).

Fig. 4. Reachability graph illustrating references in Bao et al. [2021], Wei et al. [2024], and this work. The  $\lambda$ -labeled clouds represent non-location values, and the circular nodes (○) represent store locations. All store locations are tracked (we denote tracked values using pale blue). We discuss a concrete example in Section 2.3. Despite relaxing certain internal invariants, the end-to-end guarantees remain intact (See Section 3.5).

The example in Figure 2a creates reference *outer* with *inner* as its referent, and clearly *outer* resides in a separate location. Yet in their system, *outer*’s qualifier includes *inner* (line 3), preventing references and their values from being distinct. This deep tracking model forces unnecessary dependencies [Bračevac et al. 2023], hindering optimizations such as parallelization (line 8). We refine the reference introduction rule to adopt shallow tracking, where references and referents remain distinct by default. In this refined system, *inner* and *outer* remain separate, enabling safe parallelization (Figure 2b) and allowing distinct references to share underlying objects (Figure 4c).

Adopting shallow reference tracking is non-trivial, as it weakens a key internal invariant of prior RT systems, namely that each reference qualifier conservatively captures all heap locations transitively reachable from its referent. With shallow tracking, distinct qualifiers may now reach overlapping regions of the heap, potentially breaking the separation guarantees of the system.

To reestablish separation guarantees, we develop new internal invariants to compensate for the loss of transitive reachability, demonstrating that the key reference typing rules – for assignment and dereferencing – are strong enough to enforce separation. In particular, we prove that the progress and preservation in parallel reduction corollary (Section 3.2) of Wei et al. [2024]’s system remain valid, even without explicitly tracking transitively reachable locations.

We illustrate the expressiveness of this design with additional examples in Section 2.3 and a case study on fine-grained parallelism in Section 5.2.

### 1.3 Escaping References

Sometimes the precision of the shallow model is undesirable, and a *controlled relaxation* in reference tracking is useful, especially when a mutable reference escapes its scope. Wei et al. [2024] forbids escaping references that capture resources in the escaped scope. As shown in Figure 3a, function `mkRef` returns a reference to `x`, but since `x` is local, the qualifier rules prohibit it from escaping, causing a type error. This shows the system is too strict in reference tracking and that more flexibility is needed to support escaping references.

A key challenge in supporting escaping references is that referent types are *invariant*: references cannot be covariant (allowing writing larger-than-expected values) nor contravariant (allowing reading smaller-than-expected values). To address this limitation, we introduce dual-component references of the form  $\mu x.(\text{Ref } [T^{x,q} \dots U^{x,p}])^\dagger$  with separate `write` and `read` components (see Figure 13). This formulation sidesteps the invariance challenge by splitting the referent’s qualifier into two parts, allowing us to “put in” less while “getting out” more, thus maintaining soundness.

Dual-component references enable controlled reference escaping (Section 4). In Figure 3b, before reference `c` goes out of scope, we upcast its type to the read-only cyclic reference type, *i.e.*,  $\mu z.\text{Ref } [T^{\&z}]^{c,x}$ , through reference subtyping with an escaping rule. The escaping rule ensures that the `read` qualifier of `x` is transferred to an outer scope, enabling the function to return `c` with an extended, yet sound, reachability qualifier (see `T-ESC` in Figure 13). This extension allows examples such as reference factory functions to be type checked.

Beyond escaping, dual-component references also enable general subtyping of referents (see `S-SREF` in Figure 13), allowing for finer-grained resources control, such as read-only references. In Section 5.3, we demonstrate how dual-component references with the escape rule enable new patterns of reference cell usage, further extending the expressiveness of the type system.

### 1.4 Contributions and Organization

In summary, the main contributions of this paper are as follows:

- After providing an overview of RT, we identify limitations in prior works, informally introduce the  $F_{<}^\circ$ -calculus, highlight key use cases and demonstrate expressiveness (Section 2).
- We present the formal theory and metatheory of the  $F_{<}^\circ$ -calculus, a variant of the RT systems that permits well-typed cyclic structures with a cyclic reference type and refined qualifier tracking mechanism (Section 3).
- We extend  $F_{<}^\circ$  with dual-component references and an escaping rule that allows referent qualifiers to extend beyond their defining scope while maintaining soundness (Section 4), enabling controlled imprecision in reference tracking.
- We demonstrate  $F_{<}^\circ$ ’s ability to handle well-typed cyclic references for store-based recursion (Section 5.1), shallow reference tracking for fine-grained parallelism (Section 5.2), and escaping references for reference factory functions (Section 5.3). These case studies showcase its improved precision and expressiveness over prior systems.

We discuss limitations and design trade-offs in Section 6, and related work in Section 7. All formal results in this paper have been mechanized in Rocq, with key examples directly mechanized and others derivable. They are presented using our Rocq term syntax with a standard locally nameless representation [Charguéraud 2012]. The development is available online at <https://github.com/tiakrompf/reachability> and in the extended version of this paper [Deng et al. 2025].

## 2 Reachability Types (RT)

In this section, we review the key concepts in Wei et al. [2024]’s system (Section 2.1), and informally introduce cyclic references and their role in typing examples such as Landin’s Knot (Section 2.2), shallow reference tracking (Section 2.3), and escaping references (Section 2.4).

### 2.1 Key Ideas of RT

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

```
val counter = new Ref(0) // : Ref[Int]counter  $\dashv$  [ counter: Ref[Int] $\blacklozenge$  ]
```

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

```
val counter2 = counter // : Ref[Int]counter2  $\dashv$  [ counter2: Ref[Int]counter, counter: Ref[Int] $\blacklozenge$  ]
```

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

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

```
def inc(n: Int) = { counter := !counter + n } // : inc: (Int => Unit)counter  $\dashv$  [ counter: Ref[Int] $\blacklozenge$  ]
```

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

```
def id(x: T $\blacklozenge$ ): Tx = x // : ((x: T $\blacklozenge$ ) => T(x)) $\emptyset$ 
```

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

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

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

**Reference Type: Mutable Cells.** So far, the reference type examples have only used primitive referent types (e.g. `Int`) as the referent type. Since these have untracked qualifiers ( $\emptyset$ )<sup>3</sup>, the qualifiers are elided. Wei et al. [2024]’s system supports reference types with tracked referent qualifiers, enforcing that an assigned referent must match the exact specified qualifier:

<sup>3</sup>The untracked qualifier ( $\emptyset$ ) indicates that a value has no reachable locations. Primitive values are usually untracked, since they represent pure, location-independent data.

	$e$	$y = \text{new Ref}(e)$	$!y$	$y := x$
First-Order Reference (Bao et al. [2021]) <sup>4</sup>	$e : B^\perp$	$y : \text{Ref } [B^\perp]^q$	$!y : B^\perp$	$x : B^\perp$
Nested Reference (Deep) (Wei et al. [2024]) <sup>5</sup>	$e : T^q$	$y : \text{Ref } [T^q]^{q,\blacklozenge}$	$!y : T^q$	$x : T^q$
Nested Reference (Shallow) (Section 3 in this work)	$e : T^q$	$y : \text{Ref } [T^q]^\blacklozenge$	$!y : T^q$	$x : T^q$
Cyclic Reference (Section 3 in this work)	$e : T^q$	$y : \mu x. \text{Ref } [T^{x,q}]^\blacklozenge$	$!y : T^{y,q}$	$x : T^{y,q}$
Dual-Component Reference (Section 4 in this work)	$e : T^q$	$y : \mu x. (\text{Ref } [T^{x,q} \dots U^{x,p}])^\blacklozenge$	$!y : U^{y,p}$	$x : T^{y,q}$

Fig. 5. Comparison of reference typing rules in RT. Prior works only support first-order reference typing (row 1) and nested references with transitive/deep qualifier tracking (rows 2). This work introduces shallow reference typing (row 3), cyclic reference typing (row 4), and dual-component reference typing (row 5).

```

val a = ...           // : Ta
val b = ...           // : Tb
val cell = new Ref(...) // : Ref[Ta]...
cell := a // Okay
cell := b // Error! Referent qualifier mismatch!

val a = ...           // : Ta
val b = ...           // : Tb
val cell = new Ref(...) // : Ref[Ta,b]...
cell := a // Okay, Ta <: Ta,b
cell := b // Okay, Tb <: Ta,b

```

As shown above (left), since reference `cell` has `a` as its referent qualifier, it is only permitted to be assigned a value with qualifier `a`. Assigning it with a different qualifier, e.g., `b`, results in a type error.

Qualifiers in RT can be widened to a “larger” qualifier by the qualifier subsumption rule (see `SQ-SUB` in Figure 9). For example, in the code above (right), the qualifier of the reference `cell` is adjusted to allow both `a` and `b` via qualifier subtyping. Since both  $T^a$  and  $T^b$  are subtypes of  $T^{a,b}$ , `cell` may safely be assigned either.

**Limited Reference Typing.** As shown in Figure 5, Bao et al. [2021] introduced RT that support only untracked, first-order references, where references can only refer to untracked primitive values. This means that references in their system cannot contain functions or references, since they are always tracked by the systems. Wei et al. [2024] extended their work to support nested references, where references can enclose other references. However, their system (1) lacks support for cyclic references; (2) employs imprecise aliasing tracking; (3) does not support escaping references. In the following three sections, we informally introduce our  $F_{\leq}^{\circ}$  calculus that effectively addresses the three limitations respectively.

## 2.2 Cyclic References

We introduce cyclic reference type that is of the form  $\mu z. \text{Ref } [T^q]^p$ , where  $\mu z$  creates a binder  $z$  that binds the outer qualifier  $p$ , enabling the referent qualifier  $q$  to reference its own enclosing reference. Consider the following example:

```
val outer = new Ref(...) // :  $\mu z. \text{Ref}[T^z]^{\text{outer}} + [ \text{outer} : \mu z. \text{Ref}[T^z]^\blacklozenge ]$ 
```

The code declares reference `outer` with a cyclic reference type, where the bound variable  $z$  appears in the referent qualifier. It permits assignments with values referencing itself:

```
val inner = ... // inner : Touter
```

<sup>4</sup>Bao et al. [2021] used  $\perp$  to denote untracked qualifiers, which is semantically equivalent to  $\emptyset$  in later systems. Their system does not have an equivalent concept of “freshness”. Additionally, their system uses first-order references, where referents can only have untracked primitive/base types, as denoted by  $B$  (See Figure 7).

<sup>5</sup>In this system, it is possible for references to have the  $\blacklozenge$  qualifier alone under a different context through fresh application (`T-APP-FRESH`), but it is impossible to produce separate references that encapsulate a shared value (See Section 2.3).

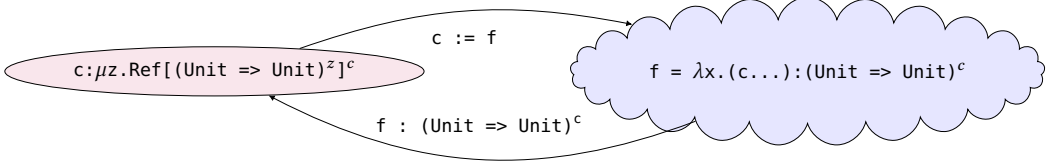


Fig. 6. Reachability graph of a well-typed **cyclic reference** assigned with a **self-capturing function**, expressing a simple cycle through the store (this work).

```
outer := inner           // Okay, because outer is a cyclic reference
```

This construct introduces a cyclic dependency: variable `inner` *reaches* reference `outer` (as indicated by its qualifier), while reference `outer`, being a cyclic reference, can *contain* `inner` via the assignment.

However, soundly incorporating cyclic dependencies into the type system is non-trivial. Consider the following naive (and erroneous) assignment rule **T-SASSGN-ERR**:<sup>6</sup>

$$\frac{\Gamma^\varphi \vdash t_1 : \mu z.\text{Ref}[T^{z,q}]^p \quad \Gamma^\varphi \vdash t_2 : T^{p,q}}{\Gamma^\varphi \vdash t_1 := t_2 : \text{Unit}^\emptyset} \quad (\text{T-SASSGN-ERR})$$

where the first premise requires that  $t_1$  has a cyclic reference type, with its referent qualifier possibly reaching either  $q$  or itself (indicated by  $z$ ). The second premise states that any term  $t_2$  can be assigned to  $t_1$ , as long as its qualifier does not exceed  $q$  and the qualifier of  $t_1$ , denoted by  $p$ .

While **T-SASSGN-ERR** provides flexibility for cyclic dependencies, it permits an assignee with *any* qualifier to be assigned to a self-reference through subtyping, violating the intended semantics of cyclic references. Consider the following example:

```
val e1 = new Ref(...) // : μx.Ref[T^x]^e1
val e2 = ...           // : T^{e2,a,b,c,d,e}
e1 := e2                // Unsound operation permitted by t-sassgn-err!
// e1: μx.Ref[T^x]^e1 <: μx.Ref[T^x]^e1,e2,a,b,c,d,e, e2: T^{e2,a,b,c,d,e} <: T^{e1,e2,a,b,c,d,e}
```

where the qualifier of the cyclic reference `e1` is incorrectly widened to accommodate an arbitrary qualifier in `e2`, while `e2` is widened to include `e1` in its qualifier. This is clearly unsound, yet it would be naively permitted by **T-SASSGN-ERR**. This issue arises because upcasting of a cyclic reference's outer qualifier allows it to generalize beyond its original scope. Thus, if assignments were based on the outer *qualifier* (as in **T-SASSGN-ERR**), the system's qualifier tracking would become inconsistent.

To prevent this unsound behavior, we restrict assignments so that the assigned reference must have the form of a *variable binder*, and the assignee's qualifier must be a *singleton qualifier*<sup>7</sup> matching that variable.

Restricting cyclic assignment to variable binders and singleton qualifiers is consistent with the reduction semantics (see Figure 10), as a variable of reference type (e.g., `e1` in this case) will always be substituted into a single location, and the assignee's singleton variable qualifier (e.g., `e2`) will be substituted into the exact same singleton location qualifier (see Section 3.2.3).

With the correct typing rule in place, the system would reject the invalid assignment operation in the previous example, thereby retaining sound qualifier tracking:

```
e1 := e2 // Error! e2's qualifier must be exactly e1
// e2: T^{e2,a,b,c,d,e} ↯ T^{e1}
```

<sup>6</sup>This rule also lacks an additional premise  $\diamond \neq q$ , which is not discussed in this section. The design decisions around this rule are explored further in Section 3.2.

<sup>7</sup>We introduce singleton qualifiers to make rules for cyclic references concise. For example, in **T-SASSGN-V** (Figure 8), we require the assignee of a cyclic reference to contain precisely a singleton variable qualifier matching that of the reference. Under static semantics, a singleton qualifier contains precisely a single variable (see Figure 7) and under dynamic semantics, it may also be a single location (see Figure 10).

**Landin’s Knot: Type Checked.** Cyclic references enable our system to type check the Landin’s knot encoding in Figure 1b. Figure 6 shows the reachability graph of the well-typed cyclic reference  $c$ , which is updated with function  $f$  that captures  $c$ . The update, *i.e.*,  $c := f$ , is type checked by **T-SASSGN-V** (in Figure 8).

When an assignment expression appears within an abstraction, where the argument is a cyclic reference, we need to make sure that the actual argument being applied to the abstraction also has a singleton variable, so that the expression after  $\beta$ -reduction is still well-typed (see **T-APP** and **T-APP-FRESH** in Figure 8).<sup>8</sup>

We present the formal typing rules for cyclic references in Section 3, and further discuss their implication and justify our design decisions in Section 6.

### 2.3 Shallow, Precise Reference Tracking

Wei et al. [2024] enforces a *saturated, deep dependency* on the reference qualifiers, where the reference’s qualifier (outer) always subsumes the referent’s qualifier (inner). This design introduces rigid coupling between the inner and the outer qualifier, leading to imprecise reachability tracking.

Consider a reference `cell` that contains an inner element  $a$ , in Wei et al. [2024]’s system:

```
val a = ... // : Ta
val cell = new Ref(a) // : (Ref[Ta])cell,a Imprecise! cell and a treated as shared
```

Reference `cell`’s qualifier overapproximates its reachable sets by including the qualifier of its enclosed element  $a$ . While such treatment is safe, it is largely imprecise and precludes complex uses of references where a more granular tracking precision is desired.

In Wei et al. [2024]’s system, one can define a function (*e.g.*, `newctx`) that attempts to circumvent outer qualifier propagation by passing a reference as a fresh argument (via **T-APP-FRESH**, see Figure 8) and via partial application, passing another fresh reference enclosing the same underlying value `inner`:

```
def newctx(c1' : Ref[Tinner]♦)(c2' : Ref[Tinner]♦) = ... // attempt to treat c1' and c2' as separate
```

However, `newctx` can never be invoked with arguments of their desired types under Wei et al. [2024]’s system, as it demands that the two separate references contain a shared value `inner`, violating the separation constraint illustrated in Figure 4b. As a failed attempt, partially applying `newctx` to `c1` results in a new function that captures `inner`, which overlaps with the `c2`. Since `c2` is required to be contextually fresh, it cannot be applied as the second argument of `newctx` (shown below to the left):

```
val inner = ... // : Tinner
val c1 = new Ref(inner) // : (Ref[Tinner])c1,inner
val c2 = new Ref(inner) // : (Ref[Tinner])c2,inner
newctx(c1)(c2) // Error! c1 overlaps c2
```

```
val inner = ... // : Tinner
val c1 = new Ref(inner) // : (Ref[Tinner])c1
val c2 = new Ref(inner) // : (Ref[Tinner])c2
newctx(c1)(c2) // Okay, c1 and c2 are disjoint
```

To address this limitation, we disentangle the reference qualifier and the referent qualifier, removing the inclusion requirement on the outer reference qualifier. As a result, the same code is correctly type checked in  $F_{<}^{\circ}$ , where the references `c1` and `c2` are disjoint, and the function `newctx` can be invoked with the desired arguments (shown above to the right).

We incorporate the shallow tracking mechanism with a novel notion of *one-step store reachability* in  $F_{<}^{\circ}$ , and present it formally in Section 3, highlighting the difference in reference introduction rule comparing to Wei et al. [2024]’s system. The shallow tracking mechanism we propose scales well to nested references (Figure 4c), where the same invariants guaranteed by the deep tracking system remain valid (Section 3.5).

<sup>8</sup>See **T-SASSGN-L** and **T-APP-VAL** in Figure 10 for the dynamic version of assignment and application.

## 2.4 Reference Escaping via Controlled Imprecision

Consider again the function `mkRef` in Figure 3a, which attempts to return a reference `c` that was created within the function body. This is not allowed in Wei et al. [2024]’s system, because `c`’s referent qualifier contains `x`, a parameter local to the function. In  $F_{<}^{\circ}$ , we introduce a controlled form of imprecision (Figure 3b).

Instead of requiring that the returned reference `c` precisely track `x`, we allow `x` to *escape* from the `read` path of its referent qualifier to its outer qualifier. In other words, we replace `x` with the binder `z`, and simultaneously add `x` to `c`’s outer qualifier. This way, the system still tracks all reachable locations on its `read` path, including the escaped qualifiers, which will be re-introduced upon dereferencing (see `T-SDEREF-DUAL` and `T-ESC` in Figure 13).

On the `write` path, we do not include the binder `z` to prevent further writes to the escaped reference. Removal of `x` from the `write` path is safe due to contravariance (see Section 4). When the write path is empty, we abbreviate `c`’s read only cyclic reference type  $\mu z.\text{Ref}[T^{\circ\dots z}]^x$  as  $\mu z.\text{Ref}[T^{\&z}]^x$  (see Figure 13).

**Read-only references via reference widening.** Building on the idea of read-only references, our system also supports read-only references through reference widening, which restricts write access while still allowing reads. For example, we can define a function that accepts a read-only reference `r` by restricting its write qualifier to  $\emptyset$ , ensuring that no tracked values can be written into it, while allowing reads:

```
def useReadOnlyRef(r:  $\mu z.\text{Ref}[\text{Ref}[\text{Int}]^{\circ\dots z}]^r$ ) =
  println(!r)           // Okay, immutable access allowed
  val a = new Ref(42)    // : Ref[Int]^a
  // !r := a            // Error! Write qualifier is empty
```

Notably, this read-only restriction applies only at the abstraction boundary, where the reference is treated as read-only upon function application through the function’s domain type. The reference itself remains mutable and may be freely updated outside the function:

```
val a = new Ref(10)      // : Ref[Int]^a
val ref = new Ref(a)    // : Ref[Ref[Int]^a]^ref
useReadOnlyRef(ref)     // Okay, reference is safely upcasted before being passed to the function
// Ref[Ref[Int]^a]^r <:  $\mu z.\text{Ref}[\text{Ref}[\text{Int}]^{\circ\dots z}]^r$ 
ref := ...              // Okay, ref is mutable outside the function
```

We formalize the dual-component reference types in Section 4 and provide examples, including read-only references for untracked values, nested escaping, and additional use cases, in Section 5.3.

## 3 $F_{<}^{\circ}$ : Reachability Types with Cyclic Reference Type

In this section, we formally present  $F_{<}^{\circ}$  that extends Wei et al. [2024]’s  $F_{<}^{\blacklozenge}$  calculus with features discussed previously. Differences from their system are highlighted in gray boxes. We also use gray to indicate typing rules in their system that are superseded by our new rules, aligning them side-by-side for easy comparison.

### 3.1 Syntax

We present the formal syntax of  $F_{<}^{\circ}$  in Figure 7. We introduce our reference type that is of the form of  $\mu x.\text{Ref}[T^q]^p$ , where  $\mu x$  binds `x` to the referent’s qualifier `q`. If  $x \in q$ , it allows the referent to refer to its reference; otherwise, it denotes a *non-cyclic reference*, and can be abbreviated as  $\text{Ref}[T^q]^p$ , which is the notation adopted from Wei et al. [2024]’s work, serving as syntactic sugar in our system. Note that if  $x \in q$ , the full  $\mu x$  notation must be used to avoid ambiguity. Due to

**Syntax**

$x, y, z$	$\in$	Var	Variables
$f, g, h$	$\in$	Var	Function Variables
$X$	$\in$	Var	Type Variables
$S, T, U, V$	$::=$	Unit   $f(x : Q) \rightarrow R$   $\mu x. \text{Ref } [Q]$   $\top$   $X$   $\forall f(X^x <: Q).Q$	Types Base Types
$B$	$::=$	Unit	Base Types
$t$	$::=$	$c$   $x$   $\lambda f(x).t$   $t t$   $\text{ref } t$   $! t$   $t := t$   $\Lambda f(X^x).t$   $t [Q]$	Terms
$p, q, r, w$	$\in$	$\mathcal{P}_{\text{fin}}(\text{Var} \uplus \{\diamond\})$	Type Qualifiers
$O, P, Q, R$	$::=$	$T^q$	Qualified Types
$\varphi$	$\in$	$\mathcal{P}_{\text{fin}}(\text{Var})$	Observations
$\Gamma$	$::=$	$\emptyset$   $\Gamma, x : Q$   $\Gamma, X^x <: Q$	Typing Environments

 $F_{<}^{\circ}$ **Reference/Qualifier Notations**

$\text{Ref } [Q]$	$::=$	$\mu x. \text{Ref } [Q]$	if $x \notin \text{fv}(Q)$	Reference Notation
$p, q$	$::=$	$p \cup q$		Qualifier Union
$q \ominus x$	$::=$	$q \setminus \{x\}$		Qualifier Exclusion
$x$	$::=$	$\{x\}$		Single Variable Qualifier
$\diamond$	$::=$	$\{\diamond\}$		Single Fresh Qualifier
$\{\text{Var}\}$	$::=$	$\{\{x\} \mid x \in \text{Var}\}$		Singleton Variable Qualifiers
$\mathcal{P}_1$	$::=$	$\{\text{Var}\}$		Singleton Qualifiers

Fig. 7. The syntax of  $F_{<}^{\circ}$ . Additions to Wei et al. [2024] are highlighted in gray boxes.

space constraints, we refer readers to our extended version [Deng et al. 2025] for the full syntax of  $F_{<}^{\circ}$ .

It is worth noting that although our  $\mu x$  notation coincides with other systems, such as recursive types [Abadi and Fiore 1996; Pierce 2002] and DOT [Amin et al. 2016; Rompf and Amin 2016], it has a distinct meaning. See Section 7 for the detailed elaboration.

**3.2 Static Typing**

As in Wei et al. [2024]'s work, the term typing judgment is written as:  $\Gamma^{\varphi} \vdash t : Q$ , as shown in Figure 8. It means that term  $t$  has type  $Q$  and is allowed to access what are observable from  $\varphi$  in typing environment  $\Gamma$ . A type  $Q$  is qualified with reachability qualifiers, and has the form  $T^q$ .

Following Wei et al. [2024], term typing assigns minimal qualifiers. **T-VAR** and **T-CST** follow their systems. In this section, we assume Unit is the only base type in the system.

**3.2.1 Reference Introduction.** **T-SREF** and **T-SREF-2** introduce the reference type for *non-cyclic* and *cyclic* references respectively. Comparing with **T-REF** in Wei et al. [2024]'s system, **T-SREF** removes the deep dependency between the inner and the outer qualifiers, by not propagating the referent qualifier (*i.e.*,  $q$ ) to its outer qualifier. Effectively, reference qualifiers in our system now denote *one-step store reachability* rather than transitive reachability, weakening the notion of reference separation and enabling more precise reference tracking. Through the revised reference introduction rules, we maintain a shallow dependency between reference and its inner referent, making reference outer qualifier track only its *one-step store reachable* resources by default.

## Term Typing

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

$$\begin{array}{c}
\frac{y : T^q \in \Gamma \quad y \in \varphi}{\Gamma^\varphi \vdash y : T^y} \quad (\text{T-VAR}) \qquad \frac{c \in B}{\Gamma^\varphi \vdash c : B^\emptyset} \quad (\text{T-CST}) \\
\frac{\Gamma^\varphi \vdash t : T^q \quad \blacklozenge \notin q}{\Gamma^\varphi \vdash \text{ref } t : (\text{Ref } T^q)^{q, \blacklozenge}} \quad (\text{T-REF}) \qquad \frac{\Gamma^\varphi \vdash t : T^q \quad \blacklozenge \notin q}{\Gamma^\varphi \vdash \text{ref } t : (\text{Ref } T^q)^{\blacklozenge}} \quad (\text{T-SREF}) \qquad \frac{\Gamma^\varphi \vdash t : T^q \quad \blacklozenge \notin q}{\Gamma^\varphi \vdash \text{ref } t : \mu x. \text{Ref } [T^{x, q}]^{\blacklozenge}} \quad (\text{T-SREF-2}) \\
\frac{\Gamma^\varphi \vdash t : (\text{Ref } T^q)^p \quad q \subseteq \varphi}{\Gamma^\varphi \vdash !t : T^q} \quad (\text{T-DEREF}) \qquad \frac{\Gamma^\varphi \vdash t : \mu x. \text{Ref } [T^q]^p \quad q \subseteq \varphi, x \quad x \notin \text{fv}(T)}{\Gamma^\varphi \vdash !t : T^{q[p/x]}} \quad (\text{T-SDEREF}) \\
\frac{\Gamma^\varphi \vdash t_1 : (\text{Ref } T^q)^p \quad \Gamma^\varphi \vdash t_2 : T^q \quad \blacklozenge \notin q}{\Gamma^\varphi \vdash t_1 := t_2 : \text{Unit}^\emptyset} \quad (\text{T-ASSGN}) \qquad \frac{\Gamma^\varphi \vdash t_1 : \mu x. \text{Ref } [T^q]^p \quad \Gamma^\varphi \vdash t_2 : T^{q \ominus x}}{\Gamma^\varphi \vdash t_1 := t_2 : \text{Unit}^\emptyset} \quad (\text{T-SASSGN}) \qquad \frac{\Gamma^\varphi \vdash y : \mu x. \text{Ref } [T^{q, x}]^p \quad \Gamma^\varphi \vdash t_2 : T^{y, q}}{\Gamma^\varphi \vdash y := t_2 : \text{Unit}^\emptyset} \quad (\text{T-SASSGN-V}) \\
\frac{\Gamma^\varphi \vdash t_1 : (f(x : T^p) \rightarrow U^r)^q \quad \Gamma^\varphi \vdash t_2 : T^p \quad \blacklozenge \notin p \quad r \subseteq \blacklozenge, \varphi, x, f \quad p \notin \mathcal{P}_1 \uplus \{\emptyset\} \Rightarrow x \notin \text{fv}(U)}{\Gamma^\varphi \vdash t_1 t_2 : U^r[p/x, q/f]} \quad (\text{T-APP}) \qquad \frac{\Gamma^\varphi \vdash t_1 : (f(x : T^{p \circledast} q) \rightarrow U^r)^q \quad \Gamma^\varphi \vdash t_2 : T^p \quad \blacklozenge \in p \Rightarrow x \notin \text{fv}(U) \quad \blacklozenge \in q \Rightarrow f \notin \text{fv}(U) \quad r \subseteq \blacklozenge, \varphi, x, f \quad p \notin \mathcal{P}_1 \uplus \{\emptyset\} \Rightarrow x \notin \text{fv}(U)}{\Gamma^\varphi \vdash t_1 t_2 : (U^r)[p/x, q/f]} \quad (\text{T-APP-FRESH}) \\
\frac{(\Gamma, f : F, x : P)^{q, x, f} \vdash t : Q \quad F = (f(x : P) \rightarrow Q)^q \quad q \subseteq \varphi}{\Gamma^\varphi \vdash \lambda f(x). t : F} \quad (\text{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 \quad q \subseteq \varphi}{\Gamma^\varphi \vdash \Lambda f(X^x). t : F} \quad (\text{T-TABS}) \qquad \frac{\Gamma^\varphi \vdash t : Q \quad \Gamma \vdash Q < : T^q \quad q \subseteq \varphi, \blacklozenge}{\Gamma^\varphi \vdash t : T^q} \quad (\text{T-SUB}) \\
\frac{\Gamma^\varphi \vdash t : (\forall f(X^x < : T^p). Q)^q \quad \blacklozenge \notin p \quad f \notin \text{fv}(U) \quad p \subseteq \varphi \quad r \subseteq \blacklozenge, \varphi, x, f \quad Q = U^r}{\Gamma^\varphi \vdash t [T^p] : Q[T^p/X^x, q/f]} \quad (\text{T-TAPP}) \qquad \frac{\Gamma^\varphi \vdash t : (\forall f(X^x < : T^{p \circledast} q). Q)^q \quad \blacklozenge \in p \Rightarrow x \notin \text{fv}(U) \quad f \notin \text{fv}(U) \quad p \subseteq \varphi \quad r \subseteq \blacklozenge, \varphi, x, f \quad Q = U^r}{\Gamma^\varphi \vdash t [T^p] : Q[T^p/X^x, q/f]} \quad (\text{T-TAPP-FRESH})
\end{array}$$

Fig. 8. Typing rules of  $F_{\leq}^\circ$ . Differences from Wei et al. [2024] are highlighted in gray boxes. The gray rules are directly taken from Wei et al. [2024] for comparison purposes, and are not a part of  $F_{\leq}^\circ$ . Note the differences in dereferencing, assignment, application rules (Section 2.2) and reference introduction rule (Section 2.3). The qualifier overlap operator ( $\circledast$ ) used in **T-APP-FRESH** and **T-TAPP-FRESH** is described in Figure 11.

Similar to Wei et al. [2024], we disallow creating references to fresh values to ensure sound reachability tracking. Without this restriction, dereferencing such a reference multiple times would produce values that, from the qualifier’s perspective, appear to be distinct fresh resources while actually referring to the same underlying resource.

**3.2.2 Dereference Rules.** **T-SDEREF** is for dereferencing references, ensuring that the retrieved value’s qualifier propagates properly. Importantly, if the referent qualifier includes the cyclic reference type binder, e.g.,  $x$ , then we substitute  $x$  with its outer qualifier  $p$  in the resulting qualifier after dereferencing. This substitution is necessary, as here  $x$  acts as an abstraction for the outer qualifier within the reference type. Once dereferencing eliminates the reference type, this abstraction must be replaced with the actual outer qualifier, as the resulting value no longer contains a reference and thus cannot retain the abstraction.

Through dereferencing, we concretize this abstraction, making the outer reference qualifier explicit in the resulting type. However, such an abstraction is limited in that the bound variables can refer only to their immediate outer qualifier, rather than qualifiers at arbitrary depth, as seen in lambda abstractions (discussed in Section 6).

For non-cyclic reference types, where the bound variable  $x$  is absent from the referent qualifier, this rule generalizes rule **T-DEREF** in Wei et al. [2024], simply retrieving the referent qualifier.

**3.2.3 Assignment Rules.** **T-SASSGN** and **T-SASSGN-V** are for safe assignment of values to references. **T-SASSGN** exhibits the same semantics as **T-ASSGN** in Wei et al. [2024]’s system when assignee is a non-cyclic reference. When the assignee is a cyclic reference, the cyclic binder is simply discarded, and a non-cyclic assignment is performed.

**T-SASSGN-V** is used for cyclic reference assignments, allowing values to reach the reference’s outer qualifier. The system permits this assignment only when the assignee is a variable and the assigned term has a singleton qualifier matching the assignee’s name. Allowing assignment without enforcing a variable-form assignee and a matching singleton qualifier, as in **T-SASSGN-ERR** (see Section 2.2), would enable assignments with arbitrary qualifiers, violating the intended type discipline. This is because **Q-SUB** allows arbitrary upcasting of a reference’s outer qualifier, and including a rule like **T-SASSGN-ERR** would lead to inconsistencies in qualifier tracking.

Why, then, is it safe to restrict assignment to term forms that are variables? By **T-VAR**, a variable is always assigned a singleton qualifier that matches its name. Although this variable’s qualifier can be upcast in an arbitrary way, the singleton qualifier is its most precise tracking prior to any upcasting. Thus, restricting the assigned term’s qualifier to a matching singleton ensures that it refers to the intended reference and cannot be widened to include other resources.

In summary, for cyclic reference assignment, we determine the assigned term’s qualifier based on the assignee’s term form, rather than its qualifier. Unlike qualifiers, which can be upcast, term forms remain fixed, ensuring correctness in assignment. Thus, **T-SASSGN-V** achieves a balance between correctness and precision while successfully supporting cyclic reference assignment.

**3.2.4 Application Rules.** Similar to Wei et al. [2024],  $F_{\leq}^{\circ}$  defines two variants of the application rule: **T-APP** for precise applications (*i.e.*, “non-fresh”) and **T-APP-FRESH** for growing applications (*i.e.*, “fresh”).

**T-APP** applies to non-fresh applications, where the argument’s qualifier is fully observable and explicitly specified. In this case, the function’s return type can depend on the function argument qualifier, and the application performs a deep, precise substitution on the argument qualifier. To ensure correct deep substitution of qualifiers in the presence of cyclic references, we impose an additional constraint: the argument qualifier must not appear in the return type, unless it belongs to a restricted set of qualifiers – empty or singleton qualifiers (denoted by  $\mathcal{P}_1 \uplus \{\emptyset\}$ ) – with which deep substitution is safe.

If the argument does not appear in the return type, type preservation is trivially maintained since no type-level substitution is required. If the argument qualifier *does* appear in the return type, it must be an empty or singleton qualifier to ensure valid deep substitution.

Why is this constraint necessary? Suppose a cyclic reference with non-empty, non-singleton qualifier is applied to an abstraction, we need to “shrink” the argument qualifier into a singleton qualifier to accommodate potential cyclic assignment to the function’s argument in the abstraction body (see Lemma 3.3). And if the argument qualifier also occurs in the function’s return type, we need to “grow” back the argument qualifier in the return type to its original form to ensure type preservation (see Section 3.5). But this is not always possible, as the argument qualifier could occur in contravariant positions of the return type.

**Subtyping**

$$\begin{array}{c}
\boxed{\Gamma \vdash q <: q} \quad \boxed{\Gamma \vdash T <: T} \quad \boxed{\Gamma \vdash Q <: Q} \\
\\
\frac{}{\Gamma \vdash B <: B} \quad (\text{s-BASE}) \qquad \frac{\Gamma \vdash S <: T \quad \Gamma \vdash T <: S \quad q \subseteq \text{dom}(\Gamma)}{\Gamma \vdash \mu z.\text{Ref } [SP]^q <: \mu z.\text{Ref } [TP]^q} \quad (\text{s-REF}) \\
\\
\frac{\Gamma \vdash T <: S \quad \Gamma \vdash S <: U}{\Gamma \vdash T <: U} \quad (\text{s-TRANS}) \qquad \frac{\Gamma \vdash P <: O \quad \Gamma, f : (f(x : O) \rightarrow Q)^\diamond, x : P \vdash Q <: R}{\Gamma \vdash f(x : O) \rightarrow Q <: f(x : P) \rightarrow R} \quad (\text{s-FUN}) \\
\\
\frac{p \subseteq q \subseteq \diamond, \text{dom}(\Gamma)}{\Gamma \vdash p <: q} \quad (\text{Q-SUB}) \qquad \frac{f : T^q \in \Gamma \quad \diamond \notin q}{\Gamma \vdash q, f <: f} \quad (\text{Q-SELF}) \qquad \frac{X^x <: T^q \in \Gamma \quad \diamond \notin q}{\Gamma \vdash p, x <: p, q} \quad (\text{Q-QVAR}) \\
\\
\frac{\Gamma \vdash q_1 <: q_2}{\Gamma \vdash p, q_1 <: p, q_2} \quad (\text{Q-CONG}) \qquad \frac{x : T^q \in \Gamma \quad \diamond \notin q}{\Gamma \vdash x <: q} \quad (\text{Q-VAR}) \qquad \frac{\Gamma \vdash p <: q \quad \Gamma \vdash q <: r}{\Gamma \vdash p <: r} \quad (\text{Q-TRANS}) \\
\\
\frac{X^x <: T^q \in \Gamma}{\Gamma \vdash X <: T} \quad (\text{s-TVAR}) \qquad \frac{}{\Gamma \vdash T <: T} \quad (\text{s-TOP}) \qquad \frac{\Gamma \vdash S <: T \quad \Gamma \vdash p <: q}{\Gamma \vdash SP <: T^q} \quad (\text{SQ-SUB})
\end{array}$$

Fig. 9. Subtyping rules of  $F_{<}^{\circ}$ .

Admittedly, enforcing the non-occurrence constraint of argument qualifier in function's return type for deep-substitution limits expressiveness, but this is necessary to preserve type soundness. We further justify this limitation in Section 6.2.

Deep substitution remains valid under the following cases, where the argument qualifier is: (1) an empty qualifier ( $p \in \{\emptyset\}$ ), or a singleton qualifier ( $p \in \mathcal{P}_1$ ).

Thus, when the argument qualifier is an empty or singleton qualifier, deep substitution in **T-APP** remains valid because the qualifier is small enough to be directly substituted using the substitution lemma (see Lemma 3.3).

Rule **T-APP-FRESH** extends the constraints of **T-APP** to fresh applications, enforcing the same non-occurrence constraint when the qualifier is not an empty or singleton qualifier. Additionally, a similar restriction applies to applications involving store locations (see Section 3.3).

**3.2.5 Type Polymorphism.** Rules **T-TAPP** and **T-TAPP-FRESH** extend the type application rule to support type and qualifier polymorphism, similar to Wei et al. [2024], where a type abstraction can be typed with **T-TABS**. Unlike function applications, type applications impose no restrictions on the argument qualifier, as types do not appear on the left-hand side of assignment operations, eliminating concerns about improper substitution.

**3.2.6 Subtyping.** Figure 9 presents the subtyping rules for  $F_{<}^{\circ}$ . Similar to Wei et al. [2024], our subtyping system distinguishes between qualifier subtyping, ordinary type subtyping, and qualified type subtyping.

Qualifier subtyping uses the standard subset inclusion (**Q-SUB**) along with two contextual reasoning rules **Q-SELF** and **Q-VAR**. Rule **Q-QVAR** introduces subtyping to qualifier variables, while **S-TVAR** enables type variable subtyping. **Q-CONG** and **Q-TRANS** ensures congruence and transitivity for qualifier subtyping.

Ordinary type subtyping includes invariance for references (**s-REF**) and standard contravariant/covariant subtyping for functions (**s-FUN**), with careful scoping of the function self-reference.

Finally, qualified type subtyping (**SQ-SUB**) decomposes into subtyping over the underlying types and their qualifiers. All subtyping rules are unchanged from prior work and are fully mechanized. We refer readers to Bao et al. [2021]; Wei et al. [2024] for the detailed explanation of these rules.

## Term Typing (with Store Typing)

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

$\ell$	$\in$	Loc	Locations
$t$	$::=$	$\dots \mid \ell$	Terms
$v$	$::=$	$\lambda f(x).t \mid c \mid \ell \mid \text{unit} \mid \Lambda f(X^x).t$	Values
$p, q, r$	$\in$	$\mathcal{P}_{\text{fin}}(\text{Var} \uplus \text{Loc} \uplus \{\diamond\})$	Qualifiers
$\varphi$	$\in$	$\mathcal{P}_{\text{fin}}(\text{Var} \uplus \text{Loc})$	Observations
$\Sigma$	$::=$	$\emptyset \mid \Sigma, \ell : \mu x.T^{q,x}$	Store Typing
$T^q$	$::=$	$\mu x.T^q \quad \text{if } x \notin q$	Store Typing Notation
$\{\text{Loc}\}$	$::=$	$\{\{\ell\} \mid \ell \in \text{Loc}\}$	Singleton Location Qualifiers
$\mathcal{P}_1$	$::=$	$\{\text{Var}\} \uplus \{\text{Loc}\}$	Singleton Qualifiers

$\frac{\Sigma(\ell) = T^q \quad q \subseteq \text{dom}(\Sigma) \quad \text{fv}(T) = \emptyset \quad \text{ftv}(T) = \emptyset \quad q, \ell \subseteq \varphi}{[\Gamma \mid \Sigma]^\varphi \vdash \ell : \text{Ref } [T^q]^\ell} \text{ (T-LOC)}$	$\frac{[\Gamma \mid \Sigma]^\varphi \vdash \ell : \mu x.\text{Ref } [T^q]^{p,\ell} \quad [\Gamma \mid \Sigma]^\varphi \vdash t_2 : T^{q,\ell} \quad \diamond \notin q \quad x \in q}{[\Gamma \mid \Sigma]^\varphi \vdash \ell := t_2 : \text{Unit}^\emptyset} \text{ (T-SASSGN-L)}$
$\frac{[\Gamma \mid \Sigma]^\varphi \vdash t_1 : (f(x : T^p) \rightarrow Q)^q \quad [\Gamma \mid \Sigma]^\varphi \vdash v : T^p \quad \Sigma(\ell) = \mu x.T^{q,x} \quad q \subseteq \text{dom}(\Sigma) \quad \text{fv}(T) = \emptyset \quad \text{ftv}(T) = \emptyset \quad q, \ell \subseteq \varphi \quad Q = U^r \quad r \subseteq \diamond, \varphi, x, f \quad p \notin \mathcal{P}_1 \uplus \{\emptyset\} \Rightarrow v \neq \ell}{[\Gamma \mid \Sigma]^\varphi \vdash t_1 v : Q[p/x, q/f]} \text{ (T-APP-VAL)}$	$\frac{\text{fv}(T) = \emptyset \quad \text{ftv}(T) = \emptyset \quad q, \ell \subseteq \varphi}{[\Gamma \mid \Sigma]^\varphi \vdash \ell : \mu x.\text{Ref } [T^{q,x}]^\ell} \text{ (T-SLOC)}$

## Well-Formed and Well-Typed Stores

$$\boxed{[\Gamma \mid \Sigma] \vdash \sigma} \quad \boxed{\Sigma \text{ ok}}$$

	$::=$	$\varphi \subseteq \text{dom}(\sigma) \subseteq \text{dom}(\Sigma) \wedge \forall \ell \in \varphi, [\Gamma \mid \Sigma]^\varphi \vdash \sigma(\ell) : \Sigma(\ell)$	
	$::=$	$[\Gamma \mid \Sigma]^{\text{dom}(\Sigma)} \vdash \sigma$	
$\frac{}{\emptyset \text{ ok}} \text{ (ST-EMP)}$	$\frac{\Sigma \text{ ok} \quad \text{fv}(T) = \emptyset \quad \text{ftv}(T) = \emptyset \quad q \in \text{dom}(\Sigma) \quad \ell \notin \text{dom}(\Sigma)}{\Sigma, \ell : T^q \text{ ok}} \text{ (ST-CON)}$		
	$\frac{\Sigma \text{ ok} \quad \text{fv}(T) = \emptyset \quad \text{ftv}(T) = \emptyset \quad q \in \text{dom}(\Sigma) \quad \ell \notin \text{dom}(\Sigma)}{\Sigma, \ell : \mu x.T^{q,x} \text{ ok}} \text{ (ST-SCON)}$		

Fig. 10. Extension with store typings for  $F_{\leq}^\circ$ . Store typing for cyclic references are highlighted in gray boxes. In contrast to Wei et al. [2024], we do not require saturated qualifiers on well-formed stores. In the dynamic semantics, singleton qualifiers ( $\mathcal{P}_1$ ) are extended to include single locations apart from single variables.

## 3.3 Dynamic Typing

The  $F_{\leq}^\circ$ -calculus follows the standard call-by-value reduction for the  $\lambda$ -calculus with mutable references. Figure 10 shows the dynamic typing rules, which incorporate term typing and subtyping with store typing  $\Sigma$ .

The reduction rules are standard, and can be found in our extended version [Deng et al. 2025].

**T-SASSGN-L** introduces a dynamic typing rule for cyclic assignments, where the assignee is a single location. This rule is the dynamic counterpart of **T-SASSGN-V**, where cyclic assignment expression stays well-typed after the reference being assigned to is substituted into a location.

**T-APP-VAL** extends typing to applications within abstraction bodies at runtime, where an argument variable is substituted with a value. As in rule **T-APP**, the empty or singleton qualifier constraint applies to argument qualifiers. Singleton qualifiers now also extend to *singleton location qualifiers*.

When the argument qualifier in **T-APP-VAL** is *not* a singleton qualifier, the argument cannot be a single location, as demanded by the substitution lemma (see Section 3.5.2). The call-by-value semantics allows us to relax the static typing constraint on the non-occurrence of the argument qualifier in the function's return type (*i.e.*,  $x \notin \text{fv}(U)$ ), as per **T-APP**. Instead, we impose a weaker

**Variable Reachability and Lookup**

$$\Gamma \vdash x \rightsquigarrow y \Leftrightarrow x : T^{q,y} \in \Gamma$$

$$\Gamma \vdash x \rightsquigarrow^* y \Leftrightarrow x \rightsquigarrow y \vee \exists z, x \rightsquigarrow z \wedge z \rightsquigarrow^* y$$

$$\Gamma \vdash x^* := \{y \mid x \rightsquigarrow^* y\}$$

$$\boxed{\Gamma \vdash x \rightsquigarrow y} \quad \boxed{\Gamma \vdash x \rightsquigarrow^* y} \quad \boxed{\Gamma \vdash x^*}$$

Variable Reachability

Variable Transitive Reachability

Variable Saturation

**Qualifier Transitive Lookup**

$$\Gamma \vdash q^0 := q$$

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

$$\Gamma \vdash q^* := q^{|\Gamma|}$$

$$\boxed{\Gamma \vdash q^n} \quad \boxed{\Gamma \vdash q^*}$$

Qualifier Transitive Lookup Base Case

Qualifier Transitive Lookup Inductive Case

Qualifier Full Transitive Lookup

**Qualifier Notations**

$$\Gamma \vdash \text{psat } q := (\bigcup_{x \in q} x^*) \subseteq q \quad \text{Propositional Qualifier Saturation}$$

$$\Gamma \vdash \text{dsat } q := q^* = q \quad \text{Deterministic Qualifier Saturation}$$

$$\Gamma \vdash p \cap q := \blacklozenge, (p^* \cap q^*) \quad \text{Qualifier Overlap}$$

$$\boxed{\Gamma \vdash \text{psat } q} \quad \boxed{\Gamma \vdash \text{dsat } q} \quad \boxed{\Gamma \vdash p \cap q}$$

Fig. 11. Qualifier Transitive Lookup. The propositional qualifier saturation predicate ( $\text{psat } q$ ) in Wei et al. [2024] is superseded by the deterministic qualifier saturation predicate ( $\text{dsat } q$ ) in the mechanization of  $F_{\leq}^{\circ}$ -calculus, which is defined in terms of the deterministic transitive lookup computation ( $q^*$ ).

requirement: the argument must not be a location value (i.e.,  $v \neq \ell$ ). This is enough because call-by-value semantics ensures that arguments are reduced to values, and the argument can have a reference type *only if* it is reduced to a location value (i.e.,  $\ell$ ) at call site.

This restriction prevents improper substitution by ensuring locations always have their corresponding singleton qualifier when applied to abstractions. In contrast, static typing must impose the non-occurrence constraint because arguments may include arbitrary terms beyond values, and their types may involve type abstractions, which cannot be checked syntactically.

Unlike Wei et al. [2024]’s  $F_{\leq}^{\bullet}$ -calculus,  $F_{\leq}^{\circ}$  drops the saturation constraint for qualifiers in the store, extending store typing to support both cyclic references and non-cyclic references.

### 3.4 Precise Transitive Closure Computation

In Wei et al. [2024]’s mechanization, the concept of transitive closure is encoded indirectly via a “saturation” predicate, which specifies that a qualifier will reach no more than itself through the corresponding store. Coupled with universal quantification, this yields an overapproximate notion of transitive closure, where a *saturated superset* of  $p$  is used to characterize  $p$ ’s transitive closure. While a precise specification of transitive closure as the *smallest saturated superset* is possible, doing so requires second-order quantification and complicates mechanization.

We replace this definition by replacing the transitive closure with a *deterministic* definition (Figure 11). By using a “fuel” measure ( $n$ ) that is set to the length of the typing context ( $|\Gamma|$ ), the computation is guaranteed to terminate while fully capturing the transitive closure of a qualifier, even when the typing context contains cycles.

We define standard monotonicity, commutativity, and distributivity properties of the transitive closure operation with respect to other operations on sets and qualifiers (e.g., splice, substitution, etc.). In our Rocq mechanization, we further show that the computational definition subsumes the saturation by proving an equivalence theorem between the two (see Lemma 3.1).

**Location Typing**

$$\boxed{\Sigma \vdash \ell : T^d}$$

$$\frac{}{\Sigma \vdash \ell : \text{Ref } [T^q]^\ell} \text{(VL-LOC)} \quad \frac{}{\Sigma \vdash \ell : \mu x.\text{Ref } [T^q]^\ell} \text{(VL-SLOC)} \quad \frac{}{\Sigma \vdash \ell : \top^d} \text{(VL-TOP)} \quad \frac{q \subseteq x, \text{dom}(\Sigma) \quad \Sigma(\ell) = T^q \quad p \subseteq q}{\Sigma \vdash \ell : \mu x.\text{Ref } [T^q]^\ell, p[\emptyset/x]} \text{(VL-STORE)}$$

**Qualifier Substitution**

$$\boxed{[\Gamma \mid \Sigma] \vdash q \xrightarrow{T} p}$$

$$\frac{\blacklozenge \not\subseteq q}{[\Gamma \mid \Sigma] \vdash q \xrightarrow{T} q} \text{(SN-EXACT)} \quad \frac{\blacklozenge \not\subseteq p \quad q \subseteq \text{dom}(\Gamma, \Sigma)}{[\Gamma \mid \Sigma] \vdash q \circledast p \xrightarrow{T} p} \text{(SN-GROW)} \quad \frac{\Sigma \vdash \ell : T^p \quad \blacklozenge \not\subseteq p \quad p \subseteq \phi}{[\Gamma \mid \Sigma] \vdash \phi \xrightarrow{T} p} \text{(SN-LOC)} \quad \frac{\Sigma \vdash \ell : T^p \quad \blacklozenge \not\subseteq \phi \quad q, \phi \subseteq \text{dom}(\Gamma, \Sigma) \quad p \subseteq \phi}{[\Gamma \mid \Sigma] \vdash q \circledast \phi \xrightarrow{T} p} \text{(SN-LOC-GROW)}$$

Fig. 12. Qualifier Substitution.

**3.5 Metatheory**

We list key lemmas and theorems that establish the soundness of  $F_{<}^\circ$ . The proof structure follows that of [Wei et al. \[2024\]](#).

**3.5.1 Saturation Lemma.**

LEMMA 3.1 (EQUIVALENCE OF DETERMINISTIC AND PROPOSITIONAL SATURATION). *For any environment  $\Gamma$  and qualifier  $q$ , we have  $\Gamma \vdash \text{psat } q \iff \Gamma \vdash \text{dsat } q$*

**3.5.2 Substitution Lemma.** We now describe qualifier substitution, a key relation that defines how qualifiers can be substituted in  $\beta$ -reduction, which leads to the substitution lemma.

Figure 12 lists the four cases where qualifier substitution can be applied. **SN-EXACT** is the case of *precise substitution* where the qualifier  $q$  is substituted for itself. This occurs for function parameters in **T-APP** or for a function's self-reference  $f$  in **T-APP** and **T-APP-FRESH**. **SN-GROW** is the case of *growing substitution* where the argument qualifier  $p$  overlaps with the function qualifier  $q$ , growing the result by  $p \setminus (\Gamma \vdash q^*)$ . **SN-LOC** is the case of *precise substitution with a location qualifier*, where a single location qualifier can be substituted similar to **SN-EXACT**. Lastly, **SN-LOC-GROW** is the case of *growing substitution with a location qualifier*, where we first grow the location qualifier  $p$  arbitrarily to some  $\phi$  and overlap it with the function the function qualifier  $q$ . similar to **SN-GROW**.

LEMMA 3.2 (SUBSTITUTION PRESERVES TRANSITIVE LOOKUP). *If  $p \subseteq \blacklozenge$ ,  $\text{dom}(\Sigma)$ , and  $[\Gamma, x : T^q \mid \Sigma] \vdash q \xrightarrow{T} p$ , then under substitution  $\theta = [p/x]$ ,  $(\Gamma \theta \vdash r\theta^*) \subseteq (\Gamma, x : T^q \vdash r^*)$ .*

PROOF. We proceed by applying transitivity of subtyping, followed by commutativity of transitive lookup under substitution, monotonicity of qualifier substitution, and environment weakening properties. The proof concludes by applying subqualifier transitivity and narrowing arguments.  $\square$

For both qualifier substitution (see Figure 12) and term substitution (Lemma 3.3), we require the location to be well-typed according to a location typing relation.

**VL-LOC** and **VL-SLOC** requires that the location qualifier of a location value  $\ell$  must be a singleton qualifier  $\ell$ , when its type is a cyclic reference type or a non-cyclic reference type respectively. **VL-TOP** allows a location to have arbitrary qualifier when its type is upcast to  $\top$ . Lastly, **VL-STORE** allows a location  $\ell$  of cyclic reference type to contain an additional qualifier  $p$  that is reachable from the store typing  $\Sigma$  at location  $\ell$ . This last case is crucial for substituting partially or fully escaped location values.

LEMMA 3.3 (TOP-LEVEL TERM SUBSTITUTION). *Suppose the following typing judgments hold:  $[x : T^q, \Gamma \mid \Sigma]^\varphi \vdash t : Q$ ,  $[\emptyset \mid \Sigma]^p \vdash v : T^p$ . Additionally, assume:  $(x : T^q, \Gamma \vdash p \circledast \varphi) \subseteq q$ , and*

$p, q \subseteq \blacklozenge$ ,  $\text{dom}(\Sigma)$ , and  $[\Gamma \mid \Sigma] \vdash q \xRightarrow{T} p$ , and  $\exists \ell, v = \ell \implies \Sigma \vdash v : p$ . Then, under substitution  $\theta = [p/x]$ , we have:  $[\Gamma\theta \mid \Sigma] \varphi^\theta \vdash t[v/x] : Q\theta$ .

PROOF. We proceed by induction on the derivation  $[x : T^q, \Gamma \mid \Sigma] \varphi \vdash t : Q$ .

In the case of **T-APP-FRESH**, the induction hypothesis requires Lemma 3.2 to establish  $(p \circledast q)\theta \subseteq p\theta \circledast q\theta$ .

In the case of **T-SASSGN-V**, we perform case analysis on the variable assignee. If the variable is the one being substituted, we use the fact that the substituted value is a reference, allowing us to conclude that it is a location. We then apply **T-SASSGN-L** to complete the case. If the variable is different from the one being substituted, the proof follows directly from the induction hypothesis. Other cases follow straightforwardly from the induction hypothesis.  $\square$

Since cyclic assignment (**T-SASSGN-V**) can occur inside an abstraction body, if the assigned term's qualifier contains the function argument qualifier, the latter must be substituted with an *empty or singleton qualifier* to ensure that the cyclic assignment remains well-typed after qualifier substitution.

As we consider only top-level, closed values in substitution, this constraint becomes relevant only when the substituted term is a location value (e.g.  $\ell$ ), as it could potentially serve as the assignee in a cyclic assignment operation. This requirement is captured by the premise  $(\exists \ell, v = \ell \rightarrow p = \ell)$ .

### 3.5.3 Main Soundness Result.

**THEOREM 3.4 (PROGRESS).** *If  $[\emptyset \mid \Sigma] \varphi \vdash t : Q$  and  $\Sigma$  ok, then either  $t$  is a value, or for any store  $\sigma$  where  $[\emptyset \mid \Sigma] \varphi \vdash \sigma$ , there exists a term  $t'$  and a store  $\sigma'$  such that  $t \mid \sigma \rightarrow t' \mid \sigma'$ .*

PROOF. By induction over the derivation  $[\emptyset \mid \Sigma] \varphi \vdash t : Q$ .  $\square$

Similar to [Wei et al. 2024], reduction preserves types up to qualifier growth:

**THEOREM 3.5 (PRESERVATION).** *If  $[\emptyset \mid \Sigma] \varphi \vdash t : T^q$ , and  $[\emptyset \mid \Sigma] \varphi \vdash \sigma$ , and  $t \mid \sigma \rightarrow t' \mid \sigma'$ , and  $\Sigma$  ok, then there exists  $\Sigma' \supseteq \Sigma$ ,  $\varphi' \supseteq \varphi \cup p$ , and  $p \subseteq \text{dom}(\Sigma' \setminus \Sigma)$  such that  $[\emptyset \mid \Sigma'] \varphi' \vdash \sigma'$  and  $[\emptyset \mid \Sigma'] \varphi' \vdash t' : T^{q[p/\blacklozenge]}$ .*

PROOF. We proceed by induction on the derivation  $[\emptyset \mid \Sigma] \varphi \vdash t : T^q$ .

For **T-APP**, we distinguish two cases:

1. If the argument qualifier is an empty or singleton qualifier, we apply the substitution lemma directly using the typing derivation from the hypothesis.

2. If the argument qualifier is larger than a singleton qualifier, we first derive the non-occurrence assumption for the argument qualifier in the function return type. This ensures that the return qualifier remains preserved after stepping. We then construct a separate typing derivation for  $t$  with a qualifier  $w$  that is more precise than  $p$ . In particular, if  $t$  is a location  $\ell$ , then  $r = \ell$ . Using this refined typing derivation, we apply the substitution lemma (Lemma 3.3), thereby completing the case.  $\square$

**COROLLARY 3.6 (PRESERVATION OF SEPARATION).** *Sequential reduction of two terms with disjoint qualifiers preserve types and disjointness:*

$$\frac{\begin{array}{l} [\emptyset \mid \Sigma]^{\text{dom}(\Sigma)} \vdash t_1 : T_1^{q_1} \quad t_1 \mid \sigma \rightarrow t'_1 \mid \sigma' \quad \emptyset \mid \Sigma \vdash \sigma \quad \Sigma \text{ ok} \\ [\emptyset \mid \Sigma]^{\text{dom}(\Sigma)} \vdash t_2 : T_2^{q_2} \quad t_2 \mid \sigma' \rightarrow t'_2 \mid \sigma'' \quad q_1 \circledast q_2 \subseteq \{\blacklozenge\} \end{array}}{\begin{array}{l} \exists p_1 p_2 \Sigma' \Sigma''. \quad [\emptyset \mid \Sigma']^{\text{dom}(\Sigma')} \vdash t'_1 : T_1^{p_1} \quad \Sigma'' \supseteq \Sigma' \supseteq \Sigma \\ [\emptyset \mid \Sigma'']^{\text{dom}(\Sigma'')} \vdash t'_2 : T_2^{p_2} \quad p_1 \circledast p_2 \subseteq \{\blacklozenge\} \end{array}}$$

PROOF. By sequential application of Preservation (Theorem 3.5) and the fact that a reduction step increases the assigned qualifier by at most a fresh new location, thus preserving disjointness.  $\square$

COROLLARY 3.7 (PROGRESS AND PRESERVATION IN PARALLEL REDUCTIONS). *Non-value expressions with disjoint observability filters can be evaluated in parallel on non-overlapping parts of the store ( $\sigma_{\uparrow\varphi}$  restricts the domain of  $\sigma$  to locations in  $\varphi$ ), and the resulting qualifiers remain separate:*

$$\frac{\begin{array}{l} [\emptyset \mid \Sigma] \varphi_1 \vdash t_1 : T_1^{q_1} \quad [\emptyset \mid \Sigma] \varphi_1 \vdash \sigma \quad t_1, t_2 \text{ non-value} \quad \Sigma \text{ ok} \\ [\emptyset \mid \Sigma] \varphi_2 \vdash t_2 : T_2^{q_2} \quad [\emptyset \mid \Sigma] \varphi_2 \vdash \sigma \quad \varphi_1 \cap \varphi_2 \subseteq \emptyset \end{array}}{\exists \sigma'_1 \sigma'_2 \Sigma_1 \Sigma_2 p_1 p_2 \varphi'_1 \varphi'_2.} \\ \begin{array}{l} t_1 \mid \sigma_{\uparrow\varphi_1} \rightarrow t'_1 \mid \sigma'_1 \quad [\emptyset \mid \Sigma_1] \varphi'_1 \vdash t'_1 : T_1^{p_1} \quad \Sigma_1 \supseteq \Sigma \\ t_2 \mid \sigma_{\uparrow\varphi_2} \rightarrow t'_2 \mid \sigma'_2 \quad [\emptyset \mid \Sigma_2] \varphi'_2 \vdash t'_2 : T_2^{p_2} \quad \Sigma_2 \supseteq \Sigma \quad p_1 \circlearrowleft p_2 \subseteq \{\diamond\} \end{array}$$

PROOF. Since  $\varphi_1$  and  $\varphi_2$  are disjoint,  $q_1$  and  $q_2$  are also disjoint. By Progress (Theorem 3.4),  $t_1$  and  $t_2$  can be reduced to  $t'_1$  and  $t'_2$ , respectively. Then by Preservation (Theorem 3.5), the contractums are well-typed. With disjoint new locations picked for the two reductions, the resulting qualifiers  $p_1$  and  $p_2$  are also disjoint.  $\square$

Our shallow semantics preserves the same strict separation guarantees as the deep model, without incurring the overhead of explicitly tracking transitively reachable locations.

### 3.6 Semantic Type Soundness

In addition to proving syntactic type soundness, we define unary logical relations for a variant of the  $F_{\leq}^{\circ}$ -calculus, and prove semantic type soundness (the fundamental property). The variant excludes type abstraction and does not include general subtyping rules, which have been modeled in prior work [Bao et al. 2025]. However, it covers the subtyping for cyclic reference types.

With our cyclic reference types, programs in our system may not terminate. Thus, we adapt Bao et al. [2025]’s semantic model by removing the termination property from the interpretation of function types and terms. We then apply the technique of step-indexed logical relations [Ahmed 2004] and define worlds indexed by execution steps. Interested readers can find our Rocq mechanization online.

## 4 Extension: Dual-Component References

This section extends reference typing in  $F_{\leq}^{\circ}$  with dual-component references, introducing separate **write** and **read** components to enable controlled imprecision and escaping. Figure 13 presents typing rules for constructing, escaping, assigning, dereferencing, and subtyping such references.

*Reference Introduction.* **T-SREF-DUAL** introduces dual-component references with separate **write** and **read** components. Both components initially share the same type and qualifier, ensuring maximum qualifier precision at allocation, but can later diverge via subtyping (**S-SREF** and **T-ESC**).

*Escaping.* **T-ESC** introduces a mechanism for controlled imprecision under the shallow reference tracking model, where reference qualifiers precisely track their own location without reaching their enclosed referents. Escaping allows inner referent qualifiers to escape into the outer reference qualifiers. Operationally, the inner **read** qualifier can partially or fully transfer its content to the outer qualifier. Under this rule, although the **read** component is covariant, it can be weakened to a *smaller* qualifier after escaping. Such weakening is permitted as long as the dropped portion is abstracted by the cyclic binder and transferred to the outer qualifier.

## Syntax

$T ::= \dots \mid \perp \mid \mu x. \text{Ref } [T^q \dots U^p]$  Types (with Bottom and Dual-Component Reference)

## Term Typing and Subtyping

$\Gamma^\varphi \vdash t : Q$     $\Gamma \vdash Q <: Q$

$$\begin{array}{c}
 \frac{\Gamma^\varphi \vdash t : T^q \quad \blacklozenge \notin q}{\Gamma^\varphi \vdash \text{ref } t : \mu x. (\text{Ref } [T^q \dots T^q])^\blacklozenge} \quad \text{(T-SREF-DUAL)} \qquad \frac{\Gamma, x : R \vdash w <: q \quad \Gamma, x : R \vdash s, p <: u, r \quad \Gamma \vdash p <: r \quad r \subseteq \varphi \quad \Gamma^\varphi \vdash t : \mu x. (\text{Ref } [T^q \dots U^s])^p}{\Gamma^\varphi \vdash t : \mu x. (\text{Ref } [T^w \dots U^{u,x}])^r} \quad \text{(T-ESC)} \\
 \\
 \frac{\Gamma^\varphi \vdash t_1 : \mu x. (\text{Ref } [T^q \dots U^s])^p \quad \Gamma^\varphi \vdash t_2 : T^{q \ominus x}}{\Gamma^\varphi \vdash t_1 := t_2 : \text{Unit}^\varnothing} \quad \text{(T-SASSGN-DUAL)} \qquad \frac{\Gamma^\varphi \vdash y : \mu x. (\text{Ref } [T^{q-x} \dots T^s])^{p,y} \quad \Gamma^\varphi \vdash t_2 : T^{q,y}}{\Gamma^\varphi \vdash y := t_2 : \text{Unit}^\varnothing} \quad \text{(T-SASSGN-V-DUAL)} \\
 \\
 \frac{\Sigma(\ell) = \mu x. T^q \quad [\Gamma \mid \Sigma]^\varphi \vdash \ell : \mu x. \text{Ref } [T^u \dots U^s]^{p,\ell} \quad w \subseteq q \ominus x \quad [\Gamma \mid \Sigma]^\varphi \vdash t_2 : T^{u\{\ell,w\}/x}}{[\Gamma \mid \Sigma]^\varphi \vdash \ell := t_2 : \text{Unit}^\varnothing} \quad \text{(T-SASSGN-L-DUAL)} \qquad \frac{\Gamma^\varphi \vdash t : \mu x. (\text{Ref } [T^q \dots U^s])^p \quad s \subseteq \varphi, x \quad x \notin \text{fv}(U)}{\Gamma^\varphi \vdash !t : U^s[p/x]} \quad \text{(T-SDEREF-DUAL)} \\
 \\
 \frac{p \subseteq \text{dom}(\Gamma)}{\Gamma \vdash \perp^p <: T^p} \quad \text{(S-BOT)} \qquad \frac{\Gamma, x : R \vdash w <: q \quad \Gamma, x : R \vdash s <: u \quad \Gamma \vdash S <: T \quad \Gamma \vdash U <: V \quad p \subseteq \text{dom}(\Gamma) \quad R = \mu x. (\text{Ref } [T^q \dots U^s])^p}{\Gamma \vdash \mu x. (\text{Ref } [T^q \dots U^s])^p <: \mu x. (\text{Ref } [S^w \dots V^u])^p} \quad \text{(S-SREF)}
 \end{array}$$

## Notations

$\mu x. \text{Ref } [Q]$	$:=$	$\mu x. \text{Ref } [Q \dots Q]$	Collapsed same qualified component
$\mu x. \text{Ref } [T^{\&p}]$	$:=$	$\mu x. \text{Ref } [T^{\varnothing \dots p}]$	Readonly references
$\mu x. \text{Ref } [T^{q \dots p}]$	$:=$	$\mu x. \text{Ref } [T^q \dots T^p]$	Collapsed same type component

Fig. 13. Typing and subtyping rules for dual-component references in  $F_{\leq}^\circ$ . The **write component** follows contravariance, while the **read component** follows covariance.

To ensure that the escaped component is properly tracked, the system adds a cyclic binder to the escaped **read** component, enabling the outer qualifier to be reintroduced upon dereferencing (See **T-SDEREF-DUAL**). Specifically, upon dereferencing an escaped reference, the resulting qualifier will contain the union of the remaining inner qualifier and the outer one, which includes all previously escaped resource. Due to contravariance, the **write** component can only shrink but not grow after escaping, similar to **S-SREF**. We do not add the cyclic binder to the **write** component, so that escaped references will not automatically allow cyclic assignment.

*Assignment.* **T-SASSGN-DUAL**, **T-SASSGN-V-DUAL**, and **T-SASSGN-L-DUAL** govern assignment to dual-component references via the contravariant **write** component. This ensures no value with a qualifier reaching more than that of its initial referent qualifier is assigned. While subtyping (**S-SREF**) may grow the **read** component, it may only shrink the **write** component. Cyclic assignment (**T-SASSGN-V-DUAL**) is permitted only for variables, mirroring **T-SASSGN-V**. In **T-SASSGN-L-DUAL**, we allow assignment to a location, similar to **T-SASSGN-L**. However, due to the introduction of dual-component references, where the **write** qualifier could have been shrunk due to subtyping prior to the assignment. We use a store lookup to recover the original referent qualifier, and permitting the assigned value to reach up to this qualifier.

*Bottom Type.* **S-BOT** introduces the bottom type  $\perp$  as a subtype of all types, preserving qualifiers. It acts as a universal placeholder for references with no write constraints. Assigning  $\perp$  to the **write**

component enforces immutability while preserving read access. This also enables safe escape of nested references by abstracting away write permissions without violating qualifier constraints.

*Dereferencing.* **T-SDEREF-DUAL** handles dereferencing by using the **read** component, which is covariant and guaranteed to retain its effective qualifier modulo any escaping. The cyclic binder in the read qualifier is substituted with the reference’s outer qualifier, as in **T-SDEREF**.

*Subtyping.* **S-SREF** enforces variance rules for dual-component references. The **write** component is contravariant, preventing larger qualifiers from being assigned; the **read** component is covariant, allowing safe expansion.

*Semantic Type Soundness.* We also prove semantic type soundness for the extension with dual-component references by adapting Bao et al. [2025]’s semantic model and applying step-indexed logical relations [Ahmed 2004]. Interested readers can find our Rocq mechanization online.

## 5 Case Studies

Our system enables reasoning about safety and separation in the presence of cyclic references, fine-grained parallelism, and controlled escapes. This section explores three case studies demonstrating its versatility.

First, we show how a fixed-point combinator (Section 5.1) can be encoded as a well-typed program. Next, we examine fine-grained parallelism (Section 5.2), where shallow reference tracking enhances concurrency. Finally, we introduce escaping, read-only references (Section 5.3), allowing references to outlive their scope while ensuring sound qualifier tracking.

### 5.1 General Fixed-Point Combinator

In this section, we present a case study demonstrating how cyclic reference types enable the implementation of a general fixpoint operator, allowing functions to define themselves recursively without explicit recursion.

Following Kiselyov [2020], we implement a general fixpoint operator through the store as follows:

```
def fix[T] (f: (g: (T -> T)♦ -> (T -> T)g) : (T -> T)♦ =
  val c = new Ref (x => x) // : μz.Ref[(T -> T)z]c
  c := f((n:T) => (!c)(n)) // : (T -> T)c
  !c
```

The key idea is to use a reference to hold the function being defined recursively, ensuring that it can be updated as needed. The fixpoint operator is polymorphic over type  $\tau$ , making it applicable to recursive computations on arbitrary data types. The function  $f$  is a higher-order function passed as an argument to the fixpoint operator, performing a “one-step” computation, using its first argument  $g$  (of type  $\tau \rightarrow \tau$ ) as the function that it refines recursively.

Unlike explicit recursive definitions, this approach enables recursion through mutable, cyclic references, allowing functions to be defined incrementally. This makes it particularly useful in settings where recursion needs to be established dynamically, such as in languages without native support for recursion or when defining mutually recursive functions.

We also implement an infinite loop and a factorial function as an instantiation of the fixpoint operator, demonstrating more practical use cases. See our extended version [Deng et al. 2025] for details.

### 5.2 Fine-Grained Parallelism

Wei et al. [2024] treats a reference and all its transitively reachable objects as shared resources. This means that if a reference `outer` points to `inner`, then modifying either one is considered an update to

the same memory region. As a result, parallel updates to `inner` and `outer` are disallowed, even if they are logically independent.

Our system removes this limitation by introducing *shallow reference tracking* via *one-step store reachability*, which allows references pointing to shared store locations to be treated as separate unless explicitly linked. This enables safe parallel updates to different parts of a data structure.

To illustrate this, we extend the previous example by introducing an additional layer of references. Now, `inner1` and `inner2` are stored inside `outer2`, which is itself referenced by `outer1`. In  $F_{<,}^{\circ}$ , `inner1`, `inner2`, `outer2`, and `outer1` are considered separate, allowing concurrent updates to each:

```
// Our system: inner1, inner2, outer2, and outer1 are separate.
val inner1 = ... // : Tinner1
val inner2 = ... // : Tinner2
val outer2 = new Ref((inner1, inner2)) // : Ref[(Tinner1, Tinner2)]outer2
val outer1 = new Ref(outer2) // : Ref[Ref[(Tinner1, Tinner2)]outer2]outer1
def par(b1: (Unit => Unit)♦) (b2: (Unit => Unit)♦) = { ... // parallelize b1 and b2 }
// Type checks: inner1, inner2, outer2, and outer1 are disjoint
par { inner1 := ... } { outer1 := ... } // any permutation is allowed
```

This approach enables *fine-grained parallelism*, where multiple threads or tasks can safely modify different parts of the structure without synchronization overhead.

### 5.3 Escaping, Read-Only References

In  $F_{<,}^{\circ}$ , we introduce a principled approach that allows escaping references while ensuring proper tracking of transitive reachability.

*Ensuring Outer Observability via Forced Escaping.* We demonstrate a case where a reference is forced to escape, ensuring that all transitively reachable locations are tracked by the outer qualifier. This recovers the deep semantics of Wei et al. [2024] via escaping.

```
// Function forcing a nested reference to escape
def escapeNestedRef(r: μz.Ref[⊥⊙...μy.Ref[⊥⊙...Ref[Int]y]z]♦) = r // Return escaped reference
val inner = new Ref(10) // : Ref[Int]inner
val mid = new Ref(inner) // : Ref[Ref[Int]inner]mid
val ref = new Ref(mid) // : Ref[Ref[Ref[Int]inner]mid]ref
escapeNestedRef(ref)
// : Ref[Ref[Ref[Int]inner]mid]ref <: μz.Ref[⊥⊙...μy.Ref[⊥⊙...Ref[Int]y]z]inner,mid,ref
// Observes all reachable locations while allowing controlled escape
```

*Strict Read-Only References via Reference Upcasting.* Comparing to the example presented in Section 2.4, a strict form of read-only reference can be enforced by setting the write component to the bottom type  $\perp$ . This ensures that the reference is immutable, even for untracked types:

```
def useImmutableRef(r: μz.Ref[⊥⊙...Ref[Int]⊙]♦) =
  !r // Allowed
// !!r := 42 // Error! Reference is fully immutable
val ref = new Ref(0)
useImmutableRef(ref) // Okay, upcasting to a fully immutable reference
// : Ref[Int]r <: μz.Ref[⊥z...Ref[Int]z]r
```

With this approach, the reference `r` is treated as fully immutable (even when modified with an untracked value, because  $\perp$  is an uninhabited type) at the abstraction boundary, while still allowing modifications in its original context.

## 6 Limitations and Future Work

This section outlines two categories of limitations. First, we discuss an intrinsic limitation of our approach to cyclic references (Section 6.1), namely the inability to form cycles across multiple “hops.” Second, we identify new restrictions on the base system (Section 6.2), focusing on constraints in deep substitution for function applications. While needed for soundness, these constraints sometimes reduce expressiveness.

### 6.1 Limitations of Our Solution

One limitation of our solution is that it cannot encode deeply nested cyclic references with multiple “hops” through the store, such as  $\mu z.\text{Ref} [\mu x.\text{Ref} [T^z]]$ . Here, the inner referent’s qualifier refers to its outer parent  $z$ , rather than the immediate parent  $x$ . This fails because our introduction rule requires cycles to tie only to the immediate parent.

Supporting such multi-hop cycles challenges well-formedness, qualifier tracking, and assignment semantics: references must stay well-scoped, qualifiers must propagate, and assignments must preserve soundness.

Possible extensions include *simultaneous reference allocation*, where mutually dependent references are introduced together, or *region-based allocation*, where references share a qualifier. Both would permit more expressive cyclic patterns while preserving soundness, which we leave for future work.

### 6.2 Limitations Introduced by Our Solution

A key limitation introduced by our approach is that deep substitution is no longer fully general in non-fresh applications (*i.e.*, **T-APP**). In prior systems, arguments could be deeply substituted without constraint, but in ours this is restricted unless the argument’s qualifier is empty or a singleton.

Specifically, in non-fresh applications, the argument must not appear in the return type. This mirrors the restriction already present in fresh applications (Section 3.2.4), and ensures sound qualifier tracking by preventing unintended aliasing.

This is not unique to our design: Capturing Types [Boruch-Gruszecki et al. 2023] impose the same restriction implicitly by requiring Monadic Normal Form (MNF), where function arguments are variables and not arbitrary expressions.

Despite the restriction, the impact on expressiveness is minor. As in DOT [Rompf and Amin 2016], precision is lost only in specific cases, and a simple workaround exists: one can introduce an intermediate variable to recover deep substitution when needed.

## 7 Related Work

**Reachability Types.** Bao et al. [2021] introduced RT that support only first-order mutable references. Wei et al. [2024] extends their work to support polymorphism and higher-order stores. We address the three key limitations in Wei et al. [2024]’s system. We introduce cyclic reference types, enabling recursive constructs without built-in mechanisms. Our system extends non-cyclic references to generalize the semantics introduced in Wei et al. [2024], providing additional flexibility for cyclic structures. We also refine the reference introduction rule to increase precision and allow flexible graph structures with nested references (see Section 2.3).

Following Dependent Object Types (DOT) [Amin et al. 2016; Rompf and Amin 2016], RT functions track their captured variables via self-references, ensuring precise tracking of shared resources. In principle, function self-references could be used within the function body to encode non-terminating recursions (*e.g.*,  $\text{def } f() = f()$ ), making recursion a built-in feature. Instead, we explicitly enable

cycles through store locations, allowing cyclic references to model recursion without having recursion as a primitive mechanism.

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

**Fixed-point Combinator.** The origin of the fixed-point combinator can be traced to computability theory [\[Rogers 1967; Sangiorgi 2009\]](#), which characterizes the computational power of recursive functions. [Bekić \[1984\]](#) formalizes recursion using fixed-point constructions, demonstrating that all definable operations arise as fixed points of monotonic functions.

Landin’s Knot [\[Landin 1964\]](#) encodes mutual recursion through self-application using mutable reference cells, a concept that later influenced cyclic references in this work. [Goldberg \[2005\]](#) generalizes Curry’s fixed-point combinator to handle variadic mutually-recursive functions. [Kise-lyov \[2020\]](#) delves into the diverse manifestations and applications of fixed-point combinators in computation. Their approach is particularly relevant to this work, as it provides a structured way to express recursion through a store using cyclic references.

Our approach to cyclic references draws inspiration from Landin’s Knot, and our case studies explore the connection between recursion and fixed points. While [Koronkevich and Bowman \[2022\]](#) restricts non-termination in higher-order reference systems via environment quantification, we deliberately enable controlled non-termination by using cyclic references to model recursion.

**Separation.** Separation Logic [\[Reynolds 2002\]](#) provides a framework for local reasoning about mutable heaps, using the separating conjunction to enforce disjointness. RT [\[Bao et al. 2021; Wei et al. 2024\]](#) adopts these principles in its application rule (**T-APP-FRESH**), where the overlap operator enforces disjointness between function and argument qualifiers. We reuse [\[Wei et al. 2024\]](#)’s notion of separation via the overlap operator (Figure 11) in **T-APP-FRESH**, but unlike prior work with universally quantified upper bounds, we compute qualifier overlap using an algorithmic transitive closure, yielding greater precision (Section 3.4).

Ensuring static separation is crucial for safe concurrent programming. Capture Separation Calculus (CSC) [\[Xu et al. 2023, 2024\]](#) extends Capture Calculus (CC) [\[Boruch-Gruszecki et al. 2023\]](#) to enforce static separation and data race freedom, ensuring non-interference of concurrently executing threads. [De Vilhena and Pottier \[2021\]](#) extend separation logic with effect handlers to reason about cooperative concurrency and mutable state, guaranteeing safe heap access.

**Recursive Types and Existential Types.** Alias Types [\[Smith et al. 2000\]](#) extends linear types with aliasing for efficient, safe memory management, leveraging existential types to abstract memory locations. [Walker and Morrisett \[2000\]](#) use existential types to capture memory shape, focusing on recursive data structures via parameterized recursive types representing circular references. [Ahmed et al. \[2007\]](#) likewise abstract memory locations with existential types, while [Grossman et al. \[2002\]](#) use them to implement closures, pairing function pointers with existentially quantified environments to enforce region bounds and liveness.

In contrast, our approach introduces a “quasi-existential” qualifier, akin to recursive self types in DOT [\[Amin et al. 2016; Rumpf and Amin 2016\]](#), where the parent reference’s identity is retained. We avoid existential types for cyclic references, since they lose this identity and reduce precision. For example, dereferencing the same cyclic reference multiple times (Section 3.2.2) would yield values that appear separate, though they come from the same reference.

**Regions, Ownership, and Substructural Type Systems.** Milano et al. [2022] introduce a system to guarantee separation and safety in concurrent programs, combining linear typing, and region-based memory management. However, while their work focuses on thread-level separation via region types, RT address higher-order functions, providing fine-grained alias tracking at the level of individual resources. Tofte and Talpin [1997] use regions to enforce sound life-time management, proving semantic equivalence between source and target semantics, ensuring the absence of use-after-free error. While their primary goal is a sound region inference algorithm for safe deallocation, our work prioritizes soundness and separation, particularly in the presence of cyclic references and recursive constructs. However, region-based guarantees such as deallocation safety can still be achieved within RT by layering an effect system [Bao et al. 2021].

Noble et al. [2023] propose local ownership in Rust, permitting multiple mutable aliases within thread-local scopes to support cycles. While this loosens Rust’s strict aliasing discipline, it relies on lexical scoping and does not statically track heap separation, which RT handles through qualifiers.

Haller and Odersky [2010] propose a capability-based system that enforces at-most-once consumption of unique references while allowing flexible borrowing. Linear Haskell [Bernardy et al. 2018] extends Haskell’s type system with linear types, providing fine-grained control over resource usage. Arvidsson et al. [2023] present Reggio, a capability-based region system enforcing isolation via a single-window-of-mutability. While effective for concurrency, its stack-like mutability discipline limits flexible aliasing across regions. RT, in contrast, supports reasoning about aliasing in the presence of pervasive higher-order functions and shared mutable state through heap reachability.

**Capabilities and Path Dependent Types, and Qualified Types.** Dependent Object Types (DOT) [Rompf and Amin 2016] is formal model of a subset of the Scala type system with proven soundness guarantee. One analogous feature of DOT with this work is that it uses term-level *recursive self types*. DOT imposes restriction on dependent application, where arguments are required to be of variable form, similar to `T-SASSGN-V` in this work (see Sections 2.2 and 3.2). However, recursive self types in DOT are used to access member types, while RT use the self-referential variable in cyclic reference for qualifier tracking. Rapoport and Lhoták [2017] extend DOT with mutable reference cells, proving the soundness of their system. However, their types remain unqualified, making it unsuitable for reachability tracking. Dort and Lhoták [2020] extend Dependent Object Types (DOT) with reference mutability system to enable fine-grained access control on references and provide immutability guarantees. Dort et al. [2024] focuses on side effect free (SEF) function types, ensuring function purity statically.

Capturing Types (CT) [Boruch-Gruszecki et al. 2023] is a type system implemented in Scala, designed to track captured capabilities. While CT makes use of boxing/unboxing operations, inspired by contextual modal type theory (CMTT) [Nanevski et al. 2008], RT’s use of the  $\blacklozenge$  marker eliminates the need of boxing and unboxing, allowing for finer-grained types that precisely model functions returning fresh values.

Qualified types have been widely adopted to perform safety analysis such as `const` qualifier inference in system-level languages [Foster et al. 2006], reference immutability in Java [Huang et al. 2012], and polymorphic type systems supporting higher-order functions [Lee and Lhoták 2023].

## 8 Conclusion

In conclusion, this work presents a variant of the reachability types system that extends its reference typing to allow cyclic references, precise and shallow referent qualifier tracking, as well as escaping of reference qualifier that enable expressive programming patterns such as fixed-point combinators, safe parallelization, and read only references.

## Acknowledgments

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

## Data Availability Statement

Rocq mechanizations can be found at <https://github.com/tiarkrompf/reachability> and Deng [2025].

## References

- Martin Abadi and Marcelo P. Fiore. 1996. Syntactic Considerations on Recursive Types. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*. IEEE Computer Society, 242–252. doi:10.1109/LICS.1996.561324
- Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007.  $L^{\wedge}3$ : A Linear Language with Locations. *Fundamenta Informaticae* 77, 4 (Jan. 2007), 397–449.
- Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State*. Princeton University.
- Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 249–272. doi:10.1007/978-3-319-30936-1\_14
- Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management. *Proc. ACM Program. Lang.* 7, OOPSLA2 (Oct. 2023), 270:1363–270:1393. doi:10.1145/3622846
- Yuyan Bao, Songlin Jia, Guannan Wei, Oliver Bračevac, and Tiark Rompf. 2025. Modeling Reachability Types with Logical Relations: Semantic Type Soundness, Termination, and Equational Theory. *Proc. ACM Program. Lang.* 9, OOPSLA (2025). doi:10.1145/3763116
- Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability Types: Tracking Aliasing and Separation in Higher-Order Functional Programs. *Proc. ACM Program. Lang.* 5, OOPSLA (Oct. 2021), 139:1–139:32. doi:10.1145/3485516
- Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. 6, 6 (1996), 579–612. doi:10.1017/S0960129500070109
- Hans Bekić. 1984. Definable Operations in General Algebras, and the Theory of Automata and Flowcharts. In *Programming Languages and Their Definition: H. Bekić (1936–1982)*, C. B. Jones (Ed.). Springer, Berlin, Heidelberg, 30–55. doi:10.1007/BFb0048939
- Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL (2018), 5:1–5:29. doi:10.1145/3158093
- Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.* 45, 4 (Nov. 2023). doi:10.1145/3618003
- Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM Program. Lang.* 7, OOPSLA2 (Oct. 2023), 236:400–236:430. doi:10.1145/3622813
- Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (Oct. 2012), 363–408. doi:10.1007/s10817-011-9225-2
- Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 15–58. doi:10.1007/978-3-642-36946-9\_3
- David G. Clarke, James Noble, and John M. Potter. 2001. Simple Ownership Types for Object Containment. In *ECOOP 2001 – Object-Oriented Programming*, Jørgen Lindskov Knudsen (Ed.). Springer, Berlin, Heidelberg, 53–76. doi:10.1007/3-540-45337-7\_4
- 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
- Karl Cray, David Walker, and Greg Morrisett. 1999. Typed Memory Management in a Calculus of Capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. Association for Computing Machinery, New York, NY, USA, 262–275. doi:10.1145/292540.292564

- Paulo Emilio De Vilhena and François Pottier. 2021. A Separation Logic for Effect Handlers. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 1–28. doi:10.1145/3434314
- Haotian Deng. 2025. Reproduction Package for Article 'Complete the Cycle: Reachability Types with Expressive Cyclic References'. Zenodo. doi:10.5281/zenodo.16995621
- Haotian Deng, Siyuan He, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2025. Complete the Cycle: Reachability Types with Expressive Cyclic References (Extended Version). arXiv:2503.07328 [cs] doi:10.48550/arXiv.2503.07328
- Vlastimil Dort and Ondřej Lhoták. 2020. Reference Mutability for DOT. In *LIPICs, Volume 166, ECOOP 2020*, Vol. 166. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 18:1–18:28. doi:10.4230/LIPICs.ECOOP.2020.18
- Vlastimil Dort, Yufeng Li, Ondřej Lhoták, and Pavel Parizek. 2024. Pure Methods for roDOT. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 13:1–13:29. doi:10.4230/LIPICs.ECOOP.2024.13
- Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. 2006. Flow-Insensitive Type Qualifiers. *ACM Transactions on Programming Languages and Systems* 28, 6 (Nov. 2006), 1035–1087. doi:10.1145/1186632.1186635
- Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102. doi:10.1016/0304-3975(87)90045-4
- Mayer Goldberg. 2005. A Variadic Extension of Curry's Fixed-Point Combinator. *Higher-Order and Symbolic Computation* 18, 3 (Dec. 2005), 371–388. doi:10.1007/s10990-005-4881-8
- Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 282–293. doi:10.1145/512529.512563
- Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer, Berlin, Heidelberg, 354–378. doi:10.1007/978-3-642-14107-2\_17
- John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. In *Proceedings of the Sixth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1991, Phoenix, Arizona, USA, October 6-11, 1991*, Andreas Paepcke (Ed.). ACM, 271–285. doi:10.1145/117954.117975
- Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. 2012. Reim & ReImInfer: Checking and Inference of Reference Immutability and Method Purity. *SIGPLAN Not.* 47, 10 (Oct. 2012), 879–896. doi:10.1145/2398857.2384680
- Songlin Jia, Guannan Wei, Siyuan He, Yuyan Bao, and Tiark Rompf. 2024. Escape with Your Self: A Solution to the Avoidance Problem with Decidable Bidirectional Typing for Reachability Types. arXiv:2404.08217 [cs] doi:10.48550/arXiv.2404.08217
- Oleg Kiselyov. 2020. Many Faces of the Fixed-Point Combinator. <https://okmij.org/ftp/Computation/fixed-point-combinators.html>.
- Paulette Koronkevich and William J Bowman. 2022. One Weird Trick to Untie Landin's Knot. (2022).
- P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (Jan. 1964), 308–320. doi:10.1093/comjnl/6.4.308
- Edward Lee and Ondřej Lhoták. 2023. Simple Reference Immutability for System F  $\llcorner$ . *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (Oct. 2023), 857–881. doi:10.1145/3622828
- Nicholas D. Matsakis and Felix S. Klock II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, Michael B. Feldman and S. Tucker Taft (Eds.). ACM, 103–104. doi:10.1145/2663171.2663188
- Mae Milano, Joshua Turcotti, and Andrew C. Myers. 2022. A Flexible Type System for Fearless Concurrency. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 458–473. doi:10.1145/3519939.3523443
- Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *ACM Transactions on Computational Logic* 9, 3 (2008), 23:1–23:49. doi:10.1145/1352582.1352591
- James Noble, Julian Mackay, and Tobias Wrigstad. 2023. Rusty Links in Local Chains\*. In *Proceedings of the 24th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '22)*. Association for Computing Machinery, New York, NY, USA, 1–3. doi:10.1145/3611096.3611097
- Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs That Alter Data Structures. In *Computer Science Logic*, Laurent Fribourg (Ed.). Springer, Berlin, Heidelberg, 1–19. doi:10.1007/3-540-44802-0\_1
- Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.
- Marianna Rapoport and Ondřej Lhoták. 2017. Mutable WadlerFest DOT. *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs* (June 2017), 1–6. doi:10.1145/3103111.3104036
- 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
- H. (Hartley) Rogers. 1967. *Theory of Recursive Functions and Effective Computability*. New York, McGraw-Hill.

- Tiark Rompf and Nada Amin. 2016. Type Soundness for Dependent Object Types (DOT). *SIGPLAN Not.* 51, 10 (Oct. 2016), 624–641. doi:10.1145/3022671.2984008
- Davide Sangiorgi. 2009. On the Origins of Bisimulation and Coinduction. *ACM Trans. Program. Lang. Syst.* 31, 4 (2009), 15:1–15:41.
- Frederick Smith, David Walker, and Greg Morrisett. 2000. Alias Types. In *Programming Languages and Systems*, Gert Smolka (Ed.). Springer, Berlin, Heidelberg, 366–381. doi:10.1007/3-540-46425-5\_24
- Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (Feb. 1997), 109–176. doi:10.1006/inco.1996.2613
- 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.
- David Walker, Karl Crary, and Greg Morrisett. 2000. Typed Memory Management via Static Capabilities. *ACM Trans. Program. Lang. Syst.* 22, 4 (July 2000), 701–771. doi:10.1145/363911.363923
- David Walker and J. Gregory Morrisett. 2000. Alias Types for Recursive Data Structures. In *Types in Compilation (Lecture Notes in Computer Science, Vol. 2071)*. Springer, 177–206.
- Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 393–424. doi:10.1145/3632856
- Yichen Xu, Aleksander Boruch-Gruszecki, and Martin Odersky. 2023. Degrees of Separation: A Flexible Type System for Data Race Prevention. *IWACO (2023)*. doi:10.48550/ARXIV.2308.07474
- Yichen Xu, Aleksander Boruch-Gruszecki, and Martin Odersky. 2024. Degrees of Separation: A Flexible Type System for Safe Concurrency. *Proc. ACM Program. Lang.* 8, OOPSLA1 (April 2024), 136:1181–136:1207. doi:10.1145/3649853

Received 2025-03-26; accepted 2025-08-12