Escape with Your Self: Sound and Expressive Bidirectional Typing with Avoidance for Reachability Types

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

Reasoning about programs in the presence of mutation and aliasing is notoriously difficult. Rust has popularized lifetime-based ownership tracking in systems programming, but its "shared XOR mutable" model is fundamentally at odds with higher-level functional programming. Reachability types offer an alternative: they enable safe sharing and escape of mutable data by tracking which resources each expression's result can reach.

To track internal reachability within complex object graphs, reachability types adopt *self-references* that let components refer to enclosing resources from inside, just like `this` pointers in OO languages. While natural for *declaratively* typing escaping data, self-references complicate subtyping and furthermore type inference: variance restricts where self-references may appear, yet useful type conversions must allow them to vary in controlled ways, which in turn imposes constraints on inference. As an undesirable result, prior works require programmers to insert term-level coercions for even just *avoidance*—avoiding ill-scoped names in types.

With all prior works being declarative, we investigate *algorithmic* reachability types in this work. We introduce a refined subtyping relation that permits more flexible usages of self-references. We further develop a sound and decidable bidirectional typing algorithm, implemented and verified in Lean. The algorithm automatically avoids ill-scoped names in types, and infers qualifiers via a lightweight unification mechanism. As a step towards practical reachability programming, we show that the system is capable of tracking diverse reachability patterns without explicit coercions in complex Church-encoded datatypes.

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

Additional Key Words and Phrases: type systems, reachability types, aliasing, bidirectional typing, avoidance

ACM Reference Format:
Songlin Jia, Guannan Wei, Siyuan He, Yuyan Bao, and Tiark Rompf. 2026. Escape with Your Self: Sound and Expressive Bidirectional Typing with Avoidance for Reachability Types. *Proc. ACM Program. Lang.* 10, PLDI, Article 257 (June 2026), 25 pages. https://doi.org/10.1145/3808335

1 Introduction

Mutable state with possible aliasing enables expressive programming patterns, but is also non-trivial to reason about, leading to memory safety violations and resource leaks. For this reason, there has been a surge of interest in language designs that regulate aliasing or mutability through a type system [11, 12, 51, 60]. Rust [33], the most notable example, has shown that lifetime tracking

Authors' Contact Information: Songlin Jia, Purdue University, West Lafayette, USA, jia137@purdue.edu; Guannan Wei, Tufts University, Medford, USA, guannan.wei@tufts.edu; Siyuan He, Purdue University, West Lafayette, USA, he662@purdue.edu; Yuyan Bao, Augusta University, Augusta, USA, yubao@augusta.edu; Tiark Rompf, Purdue University, West Lafayette, USA, tiark@purdue.edu.

ACM 2475-1421/2026/6-ART257
https://doi.org/10.1145/3808335

based on ownership types is an eminently practical way of ensuring memory safety in a low-level imperative system language. To realize a similar flavor of lifetime reasoning in higher-order languages, however, adapting Rust's approach would be restrictive: its "*shared XOR mutable*" model—permitting a resource to be either shared or mutable, but not both—prohibits many functional programming idioms that require capturing and sharing mutable values.

Reachability types [5, 6, 16, 53] are a recent proposal to bring the benefits of lifetime reasoning and principled aliasing to higher-level languages. The key idea is to augment the type system with a qualifier that tracks which resources an expression's result can reach. Reachability types impose restrictions on sharing and separation at a fine-grained, per-expression level, rather than as global ownership invariants. This permits mutable data to be shared or to escape its defining scope while retaining static safety guarantees, even in the presence of higher-order functions and polymorphic types. Below, we exemplify the programming patterns this approach enables, including those beyond the *shared XOR mutable* model.

Controlling Lifetimes. Reachability types naturally support resource-management patterns where a handle must remain scoped to its defining context. Consider a `withFile` combinator that opens a file, passes the handle to a callback, and closes it upon return. In the type of `withFile`, the freshness marker ♦ signifies that the file handle is new to the callback and not reachable in other ways:

```
// withFile: ∀[A^a].(path: String) → (body: File♦ → A^a) → A^a
withFile[Unit]("log.txt")(fun f => write(f)("hello") )
```

Soundness requires that the file handle not escape its defining scope [36, 40, 54], and the type of the callback `body` enforces this: the argument `f` is a fresh file handle and lies outside the domain of the return type A^a, so no instantiation of A^a for `withFile` can allow a call leaking `f`:

```
let f = withFile[File♦]("log.txt")(fun f => f )   // Error: f not allowed in result type
```

This guarantee is analogous to Rust's lifetime-based scoping using higher-ranked trait bounds (HRTB). The two systems enforce the same invariant here, but diverge when sharing is introduced.

Sharing and Separation. A useful pattern is to define several writers over the same file handle. For example, consider loggers at different severity levels:

```
withFile[Unit]("log.txt")(fun f =>
  let warn  = fun (msg: String) => write(f)("WARN: " ++ msg)   // warn:  (String => Unit)^f
  let error = fun (msg: String) => write(f)("ERROR: " ++ msg)  // error: (String => Unit)^f
  warn("low memory"); error("disk full") )
```

Reachability types track that both `warn` and `error` reach the handle `f`, allowing them to safely coexist. This tracking enables static enforcement of separation: in the type of a parallel combinator `par` [5, 53], freshness on both parameters requires that each not be reachable from the other. The type system accepts the call when qualifiers are disjoint and rejects it when they overlap, for example:

```
// par: (f: (Unit → Unit)♦) → (g: (Unit → Unit)♦) → Unit, requires f and g separate
par(fun _ => warn("starting") )(fun _ => error("failed") )     // Error: both reach f
```

In Rust, a mutable resource cannot be borrowed twice, so the pattern above requires falling back to dynamic reference-counting, *e.g.*, using `Rc<RefCell<T>>` or `Arc<Mutex<T>>`, erasing the aliasing relationship from the types of `warn` and `error`. Guarding against conflicting borrows then requires runtime mechanisms.

Into Stateful Objects. Sharing and separation reasoning extends to data structures that capture resources. In a functional setting, objects can be naturally modeled as products of closures with shared state, making it essential to track their common, hidden resources. The example below encodes a `Counter` class [53, 55] with a hidden state `c` and two mutation methods. Reachability types track that both methods access `c` without referring to `c` outside its defining scope. Instead, a

self-reference p is introduced on the result pair using the μ-notation, replacing c in the qualifiers of the two closures:

```
let Counter = fun (n: Int) =>                  //: Int → μp.Pair[(Unit → Unit)^p, (Unit → Unit)^p]^♦
  let c = ref n
  Pair(fun () => c := !c + 1, fun () => c := !c - 1)  //: Pair[(Unit → Unit)^c, (Unit → Unit)^c]^c
```

When the returned pair is bound to the variable ctr, its projected components are then understood to reach ctr as well, thus preserving the sharing invariant:

```
let ctr = Counter(0)                           // ctr: Pair[(Unit → Unit)^ctr, (Unit → Unit)^ctr]^ctr
let incr = fst(ctr); let decr = snd(ctr)
par(fun _ => incr() )(fun _ => decr() )        // Error: both methods reach ctr
```

Rust would require dynamic reference-counting for such a case (cf. Figure 1b, Bao et al. [6]).

From Self-References to Algorithms. While reachability types have shown great promise for expressiveness, practical type checking and inference algorithms have not been addressed: prior works focused on declarative formulations and proofs of type soundness, with no algorithmic counterpart. The main source of difficulty is *self-references*—qualifiers that let components refer to their enclosing resources (like **this** pointers), adopted from path-dependent types in DOT [2, 47]. As seen in the Counter example, self-references are essential for tracking escaping data, but they complicate both the subtyping needed for *avoidance*[1] and the inference of qualifiers (see Section 2.2).

Prior work [5, 16, 53] treats types and qualifiers separately in subtyping, leaving reachability opaque and self-references inert; avoidance thus requires term-level coercions. Challenging their assumption that *the locations a value may reach are fixed* motivates a refined semantic model in which they depend on the type assigned to it, a combined type-and-qualifier subtyping that enables avoidance conversions involving self-references directly in subtyping, and algorithmic procedures for automated qualifier inference and avoidance built atop this new foundation.

Contributions and Organization. We address the open challenges of type checking/inference with avoidance for reachability types [5, 16, 53] by (1) proposing the $G^{\blacklozenge}_{<:}$-calculus with combined type-and-qualifier subtyping, and (2) developing a sound and decidable bidirectional typing algorithm $G^{\blacklozenge}_{\Leftrightarrow}$ to the specification of $G^{\blacklozenge}_{<:}$, both fully mechanized in Lean. Together, these systems enable end-to-end support for expressive high-level constructs with only modest annotation overhead.

- We review the core concept of reachability types, and identify key challenges in algorithm design on avoidance and qualifier inference (Section 2), motivating our design of $G^{\blacklozenge}_{<:}$ and $G^{\blacklozenge}_{\Leftrightarrow}$.
- We introduce the $G^{\blacklozenge}_{<:}$-calculus (Section 3), featuring type-and-qualifier subtyping and qualifier holes in algorithmic contexts. These mechanisms enable flexible conversions involving self-references and allow type assignment under partially specified function qualifiers.
- We develop the bidirectional typing algorithm $G^{\blacklozenge}_{\Leftrightarrow}$ (Section 4), with mechanized proofs of soundness and decidability relative to $G^{\blacklozenge}_{<:}$. The algorithm infers qualifiers for expressions and avoids ill-scoped qualifiers in typing while tracking escaping data in higher-order settings.
- We evaluate $G^{\blacklozenge}_{\Leftrightarrow}$ on programs using higher-order functions and data structures (Section 5), demonstrating expressiveness for resource and lifetime reasoning, with moderate annotation and performance overhead. We further discuss alternative designs and possible extensions.

We discuss related work in Section 6 and conclude the paper with Section 7. In our artifact [28], we provide the mechanization of $G^{\blacklozenge}_{<:}$ and $G^{\blacklozenge}_{\Leftrightarrow}$, and examples for empirical evaluation.[2]

[1] Avoidance converts types to remove variables about to go out of scope. In the Counter example, this means replacing c with the self-reference *p* when the pair is returned.
[2] Also available at https://github.com/TiarkRompf/reachability/tree/main/checking/lean_v2.

2 Motivation

Reachability types [6, 16, 53] concern the use of resources in impure functional languages. They track resources by *reachability qualifiers* and enable functions to constrain their arguments by sharing and separation. While prior works *declaratively* characterize the valid type assignments of terms, they do not spell out *algorithmic* steps for checking such assignments, nor do they support inferring types and qualifiers for ergonomic programming. In this section, we present an informal overview of our reachability type system $G^{\blacklozenge}_{\Leftrightarrow}$ with bidirectional typing and avoidance support.

2.1 Elements of Reachability Qualifiers

While exact resource identities—often memory locations—are unavailable at compile time, reachability types [5, 53] approximate them statically using (1) *variables* for resources with externally visible names, (2) *freshness* for unnamed resources, *e.g.*, new allocations, and (3) *self-references* for components of unnamed resources to refer to their enclosing resources or themselves.

Variables. When the new allocation `ref 42` is assigned to the variable `b`, we infer ($\Rightarrow$) all later uses of `b` to be $\texttt{Ref[Int]}^{\texttt{b}}$,[3] where the qualifier `b` signifies that values resulting from the expression `b` reach no more resources than the established variable `b`:

```
let b = ref 42; b            // [b: Ref[Int]♦] ⊢ b      ⇒ Ref[Int]^b
```

Qualifiers of functions include their free variables, like `b` in the example below, reflecting the fact that at runtime, the assignments of such free variables are recorded (thus reached) in the closure:

```
fun () => b := !b + 1        // [b: Ref[Int]♦] ⊢ <fun>  ⇒ (Unit → Unit)^b
```

Freshness. Expressions marked fresh $\blacklozenge$ represent the resources *unreachable* from existing variables, so that operating on fresh values causes no interference. New allocations by `ref` are always fresh. Below, both `ref 42` (later reached by `b`) and `ref 43` yield fresh references, guaranteed to be separate:

```
let b = ref 42; ref 43       // [b: Ref[Int]♦] ⊢ ref 43  ⇒ Ref[Int]♦    ← separate from b
```

Freshness markers in argument qualifiers represent *contextual* freshness: they are unreachable from variables reached in the function, but may overlap with variables not observed by the function. Below, `accumulate` takes a fresh argument `x`. Thus, parameters to `accumulate` should be separate from its captured values, specifically `acc`, but may still reach other variables:

```
let b = ref 42; let acc = ref 0
let accumulate =
  fun (x: Ref[Int]♦) =>      // [···, acc: Ref[Int]♦] ⊢ <fun>  ⇒ ((x: Ref[Int]♦) → Unit)^acc
    acc := !acc + !x
accumulate(b)                // Okay:  'b' not observed by 'accumulate'
accumulate(acc)              // Error: 'acc' not separate from 'accumulate'
```

Self-References. Just like `this` pointers in object-oriented languages allowing fields to be reached by methods, self-references allow resources to be reached from inside, which is crucial to representing escaping data structures. Below, we model an object `obj` with a mutable state and a `Pair` of `getter`/`setter` methods. While the name `b` is not visible outside, we still need to track that the methods reach shared resources. As the `Pair` becomes the new logical owner of the reference `b`, we use its self-reference `p` introduced by the μ-notation to qualify the pair components:

```
let obj =                    // [···, obj: ···♦] ⊢ obj  ⇒ μp.Pair[(Unit → Int)^p, (Int → Unit)^p]^obj
  { let b = ref 42
    Pair(fun () => !b, fun (n: Int) => b := n) }
let getter = fst(obj)        // [···, obj: ···♦] ⊢ fst(obj)  ⇒ (Unit → Int)^obj   // p mapped to obj
let setter = snd(obj)        // [···, obj: ···♦] ⊢ snd(obj)  ⇒ (Int → Unit)^obj
```

[3] As a convention, we use purple to emphasize results inferred by the algorithm.

2.2 Challenges for Algorithmic Reachability Types

Reachability types refer to term variables in types, making them dependent. However, without term evaluation in types, they are faced with the *avoidance problem* known in bounded existential types [26], DOT [2, 47], and module systems [7, 20, 31, 32, 48]: they need mechanisms to remove names that are about to go out of scope, *e.g.*, avoiding `b` in the `obj` example above.

To deal with avoidance, reachability types allow (1) substituting variables with qualifiers and (2) converting types using self-references. Nevertheless, both mechanisms are nontrivial for inference algorithms. We analyze these mechanisms and their challenges in the rest of this section.

Substituting Variables. As a minimal example of dependent functions, `identity` returns its argument `x` and captures no free variable. Its result type reaches the bound variable `x`:

```
let identity = fun (x: Ref[Int]♦) => x  // [···    ] ⊢ <fun> ⇒ ((x: Ref[Int]♦) → Ref[Int]^x)^∅
```

After function application, `x` is not defined in the scope and thus can no longer occur. To preserve reachability tracking, we can substitute `x` in the type with parameter qualifiers, `b` or ♦, respectively:

```
let b = ref 42; identity(b)  // [···, b: Ref[Int]♦] ⊢ <app> ⇒ Ref[Int]^b      // Ref[Int]^x [b/x]
identity(ref 42)             // [···               ] ⊢ <app> ⇒ Ref[Int]♦      // Ref[Int]^x [♦/x]
```

Such *dependent application* achieves lightweight reachability polymorphism [53] ergonomically via function parameters [8, 49], in addition to explicit polymorphism via separate quantification.

Variables bound by `let` can be substituted similarly. Below, we replace `b` by ♦ in the result type:

```
let b = ref 42               // [···               ] ⊢ <let> ⇒ Ref[Int]♦      // Ref[Int]^b [♦/b]
b                            // [···, b: Ref[Int]♦] ⊢ b     ⇒ Ref[Int]^b
```

Avoidance Conversion by Self-References. Substitution with freshness is restricted within types. Take *the escaping closure* below for example: the function expression returns the captured reference `b`. When typing the overall let binding, the bound variable `b` can no longer occur in the resulting type, but substituting both `b`'s with ♦ would change the type to mean returning new references:

```
let b = ref 42               // [···               ] ⊢ <let> cannot be (Unit → Ref[Int]♦)♦
fun () => b                  // [···, b: Ref[Int]♦] ⊢ <fun> ⇒       (Unit → Ref[Int]^b)^b
```

To prevent such an unintended change, we note that substitution with ♦ cannot take place inside types: it is the substitution in the function return qualifier that causes the change.

To avoid the internal occurrence of `b` without resorting to substitution, we approximate it using self-references. We convert ($\ll$) the type of the `let`-body to use the self-references `f`,[4] so that substituting `b` is only required for the top-level qualifier:

```
let b = ref 42               // [···               ] ⊢ <let> ⇒       (f(Unit) → Ref[Int]^f)♦
fun () => b                  // [···, b: Ref[Int]♦] ⊢ <fun> ⇒ ··· ≪ (f(Unit) → Ref[Int]^f)^b
```

To summarize, substitutions on bound variables are subject to the restriction that they must be *either non-fresh, or non-deep*. This restriction is already seen in prior work [16, 53] from their declarative systems. Algorithmically, when a deep occurrence needs substitution involving freshness, it has to be first removed by avoidance conversions using self-references.

Algorithmic Challenges. Both mechanisms are nontrivial to implement. Substitution requires *inferring precise parameter qualifiers* for precise substitution results. Avoidance conversion requires devising a *type conversion scheme* that removes undesired variables and is sound with respect to a subtyping relation.

2.2.1 *Inferring Qualifiers for Parameters.*

In bidirectional typing systems [21] without reachability, when applying a function like `identity`, the parameter is checked ($\Leftarrow$) against the argument type:

```
let b = ref 42; identity(b)  // [···, identity: Ref[Int] → Ref[Int], ···] ⊢ b ⇐ Ref[Int]
```

[4] In the function type `f(Unit)` $\rightarrow \cdots$, we define `f` as its self-reference. We omit μ-notations for functions.

For reachability types, we need to adopt a hybrid checking/inference mode ($\Leftarrow\ ^{\Rightarrow}$) similar to that seen in refinement types [44], checking the type but inferring the qualifiers:

```
let b = ref 42; identity(b)  //[···,identity:(x:Ref[Int]♦)→ Ref[Int]ˣ,···] ⊢ b ⇐ Ref[Int]⇒ᵇ
```

Although inferring `b` in `identity(b)` is straightforward, it gets more complicated when self-references are involved in argument types, which introduces constraints from typing the parameter.

Constraints from Self-References. Below, `callGet` calls its argument `get` to retrieve what it captures:

```
let callGet =  fun (get: ( g(Unit) → Ref[Int]ᵍ )♦) => get() //: Ref[Int]^get
```

With the result of `get` qualified by its self-reference `g` and the result of `callGet` qualified by `get`, both results may reach the same resource as `get`. Such constraints among reachability can be alternatively understood using an explicit qualifier quantification, illustrated below as `callGet2`:

```
let callGet2 = fun [q <: ♦](get: (Unit → Ref[Int]^q)^q) => get() //: Ref[Int]^q
```

Invoking `callGet2` requires instantiating the qualifier variable `q` to satisfy the typing:

```
let b = ref 42; callGet2[b](fun _ => b)  //[···,b:Ref[Int]♦] ⊢ <fun> ⇐ (Unit → Ref[Int]^q)^q [b/q]
```

This is analogous to instantiating type variables for type polymorphism, whose inference is known to be nontrivial [14, 22, 38, 43, 57, 58]. While $G^{\blacklozenge}_{\Leftrightarrow}$ does not deal with type instantiations, it infers parameter qualifiers for invoking `callGet`, achieving similar expressiveness ergonomically:

```
let b = ref 42; callGet(fun _ => b)      //[···,b:Ref[Int]♦] ⊢ <fun> ⇐ (g(Unit) → Ref[Int]ᵍ)⇒ᵇ
```

We elaborate on our approach to qualifier inference in Section 2.5.

2.2.2 Self-References Conversions for Avoidance. As analyzed earlier, typing the *escaping closure example* requires the following type conversion to remove the variable `b` from inside the type:

$$[\cdots, \mathtt{b}:\mathtt{Ref[Int]}^{\blacklozenge}] \ \vdash\ (\mathtt{Unit} \to \mathtt{Ref[Int]}^{\mathtt{b}})^{\mathtt{b}} \ \ll\ (\mathtt{f(Unit)} \to \mathtt{Ref[Int]}^{\mathtt{f}})^{\mathtt{b}}$$

With `b` included in the function qualifier, the self-reference `f` then replaces `b` in the result qualifier. For such conversions, we need to ensure that they are sound, and that they can be applied generally.

Soundness. Prior work [16, 53] could not justify the conversion resulting in a supertype, largely due to the fact that their subtyping relations (<:) are designed without top-level qualifiers:

$$[\cdots, \mathtt{b}:\mathtt{Ref[Int]}^{\blacklozenge}] \ \vdash\ \mathtt{Unit} \to \mathtt{Ref[Int]}^{\mathtt{b}} \ \not<:\ \mathtt{f(Unit)} \to \mathtt{Ref[Int]}^{\mathtt{f}}$$

Unable to see that *the function qualifier includes* `b` in subtyping, prior works require η-expanding escaping closures for *retyping* them. Manifested in programming with data structures like pairs, this necessitates term-level coercions like `conv` below,[5] making it unideal for both theory and practice:

```
def conv[Aᵃ, Bᵇ](p: Pair[Aᵃ, Bᵇ]{a,b,♦}): μp.OPair[Aᵖ, Bᵖ]{a,b} =
  OPair(fst(p), snd(p))  // reconstruct Pair to use self-references in types (OPair)
```

To justify seamless type conversions, we thus need to improve the notion of subtyping. More than just reinstating missing qualifiers, we need to deal with the variance of self-references.

Generalizing for Variance. The *escaping closure example* requires avoidance in function result qualifiers, which is the only position where prior works [16, 53] allow a self-reference. In contrast, variables about to go out of scope may occur in places with different variances and depths:

```
let b = ref 42                    // fresh 'b' about to go out of scope
fun () => b                       // covariant:      Unit → Ref[Int]ᵇ
fun (x: Ref[Int]ᵇ) => !x          // contravariant:  Ref[Int]ᵇ → Int
ref b                             // invariant:      Ref[Ref[Int]ᵇ]
fun (f: Ref[Int]ᵇ → Int) => f(b)  // deep covariant: (Ref[Int]ᵇ → Int) → Int
```

Generally avoiding them requires a systematic scheme to use self-references. We discuss our avoidance algorithm in Section 2.3, and its soundness foundation—our new subtyping—in Section 2.4.

[5] Adapted from Section 8.1, Wei et al. [53]. We elaborate further in Section 5.1.

2.3 Self-Reference Conversions for Avoidance

At a high-level, our algorithmic avoidance involves replacing variables in covariant positions by self-references and simply removing the contravariant ones. We detail our solution as follows.

Covariant Occurrences. For the base case from Section 2.2.2, we replace `b` in the covariant result qualifier with the self-reference `f`, and we add `b` in the function qualifier if it is not yet included:

```
let fn = { let b = ref 42    // fn     ⇒ (f(Unit) → Ref[Int]^f)^fn
           fun () => b     } // <fun>  ⇒ (Unit → Ref[Int]^b)^b  ≪  (f(Unit) → Ref[Int]^f)^b,b
```

When the escaping closure is later named `fn` and applied, the self-reference `f` becomes another bound variable needing substitution. We replace it with the qualifier of `fn`, *i.e.*, `fn` itself:

```
fn()                          // <app>  ⇒ Ref[Int]^fn                    // Ref[Int]^f [fn/f]
```

Contravariant Occurrences. When the unwanted variable occurs in contravariant positions, we remove it, analogous to replacing ill-scoped type variables with the bottom type in algorithmic System $F_{<:}$ [43]. Illustrated below, we change the argument type from `Ref[Int]^b` to `Ref[Int]^∅`:

```
let fn = { let b = ref 42
           fun (x: Ref[Int]^b) => !x }  // <fun>  ⇒ (Ref[Int]^b → Int)^b  ≪  (Ref[Int]^∅ → Int)^b
```

Such a conversion renders `fn` non-callable, but it is necessary in this specific example: the function is originally defined to *receive no more references than* `b`, and there is no qualifier other than `b` itself that can keep this invariant. In practice, a dummy function type like that of `fn` should be a signal for an overly conservative type annotation somewhere.

Invariant Occurrences. Aligned with recent development [16, 24], we adopt the notion of *dual-component* references, with a contravariant *put* type and a covariant *get* type; types with a single invariant referent are then seen as shorthands for two components being the same. To avoid `b` in the example below, we consider the two components separately: we remove `b` in the *put* qualifier, replace `b` with the self-reference `h` in the *get* qualifier, and add `b` to the top-level reference qualifier:

```
let r = { let b = ref 42     // r      ⇒ μh.Ref[Ref[Int]^∅..Ref[Int]^h]^r
          ref b          }   // ref b  ⇒ Ref[Ref[Int]^b]^♦  ≪  μh.Ref[Ref[Int]^∅..Ref[Int]^h]^♦,b
```

To read the reference `r` resulting from escaping, similar to when calling functions, we need to replace `h` with the actual qualifier `r`, signifying that the extracted result is internal to `r`:

```
!r                            // !r     ⇒ Ref[Int]^r                     // Ref[Int]^h [r/h]
```

By necessity, `r` is made *read-only* with the put qualifier ∅ to accept *no more assignment than* `b`.

Deep Occurrences. Deep, covariant uses of the unwanted variable are replaced with the self-reference of the outermost function/reference. Exemplified below, we replace `b` inside the argument type with `f`. To apply this escaped `fn`, `f` in the argument type should first be replaced with `fn`:

```
let fn = { let b = ref 42   // <fun> ⇒ ((Ref[Int]^b → Int) → Int)^b ≪ (f(Ref[Int]^f → Int) → Int)^b,b
           fun (get: Ref[Int]^b → Int) => get(b) }
fn(fun b => !b)             // <fun> ⇐ (Ref[Int]^fn → Int)  ⇒^fn     // (Ref[Int]^f → Int) [fn/f]
```

2.4 Type-and-Qualifier Subtyping for Avoidance

To justify avoidance conversions, we propose a combined form of type-and-qualifier subtyping, with self-references enabled in covariant positions for both expressiveness and soundness.

Combined Subtyping. In our declarative specification $G^{\blacklozenge}_{<:}$, we use the symbol $\leqslant:$ for the extended subtyping on qualified types. The base covariant conversion case can be formalized as:

$$[\cdots, \mathtt{b} : \mathtt{Ref[Int]}^{\blacklozenge}] \;\vdash\; (\mathtt{Unit} \to \mathtt{Ref[Int]}^{\mathtt{b}})^{\mathtt{b}} \;\leqslant:\; (\mathtt{f(Unit)} \to \mathtt{Ref[Int]}^{\mathtt{f}})^{\mathtt{b}}$$

With both sides agreeing on the outermost qualifier `b`, this fact enables proving the subtyping relation, *packing* variable names into self-references. $G^{\blacklozenge}_{<:}$ can also justify *unpacking* that returns self-references into variables, making post-avoidance types compatible with operations before:

$$[\cdots, \texttt{fn} : (\texttt{f(Unit)} \to \texttt{Ref[Int]}^{\texttt{f}})^{\blacklozenge}] \quad \vdash \quad (\texttt{f(Unit)} \to \texttt{Ref[Int]}^{\texttt{f}})^{\texttt{fn}} \quad \leqslant: \quad (\texttt{Unit} \to \texttt{Ref[Int]}^{\texttt{fn}})^{\texttt{fn}}$$

Supporting Growing Qualifiers. Qualifiers in subtyping may not fully agree, for example:

$$[\cdots, \texttt{a} : \texttt{Ref[Int]}^{\blacklozenge}, \texttt{b} : \texttt{Ref[Int]}^{\blacklozenge}] \quad \vdash \quad (\texttt{Unit} \to \texttt{Ref[Int]}^{\texttt{b}})^{\texttt{a}} \quad \leqslant: \quad (\texttt{f(Unit)} \to \texttt{Ref[Int]}^{\texttt{f}})^{\texttt{a,b}}$$

In $G^{\blacklozenge}_{<:}$, this is still a valid subtyping relation, but requires a transitivity step changing the qualifier of the subtype (left) from `a` to `a,b`, via the type $(\texttt{Unit} \to \texttt{Ref[Int]}^{\texttt{b}})^{\texttt{a,b}}$; it is then upon the typing algorithm to find such necessary intermediate steps to justify subtyping relations (see Section 2.5).

Moreover, with qualifiers growing in subtyping relations, self-references in the two sides may carry differing interpretations. Illustrated below, dictated by function qualifiers, the self-reference `f` in the subtype (left) can reach at most `a`, but in the supertype (right) it may additionally reach `b`:

$$[\cdots, \texttt{a} : \texttt{Ref[Int]}^{\blacklozenge}, \texttt{b} : \texttt{Ref[Int]}^{\blacklozenge}] \quad \vdash \quad (\texttt{f(Unit)} \to \texttt{Ref[Int]}^{\texttt{f}})^{\texttt{a}} \quad \leqslant: \quad (\texttt{f(Unit)} \to \texttt{Ref[Int]}^{\texttt{f}})^{\texttt{a,b}}$$

Thus, in subtyping, self-references are *covariant*. In $G^{\blacklozenge}_{<:}$, we simply require that self-references occur only in covariant positions in types. This aligns with our algorithmic avoidance scheme, which never involves self-references contravariantly.

2.5 Inferring Function Qualifiers

Inference of qualifiers is most challenging for functions. A function's qualifier is determined primarily by (1) *observations* collected from the function body, and (2) constraints induced by subtyping relations that involve self-references. Collecting observations is necessary for new function definitions, whereas resolving constraints is required when we use existing functions in different types. We detail the two sources as follows.

Collecting Observations. When evaluating expressions, resources may be transiently involved without necessarily being reached in the resulting value. Such *observations* naturally include free variables of expressions. In the example below, the expression `!b` observes `b`, as marked beside the context on the right:

```
let b = ref 42; !b     // [···, b: Ref[Int]♦]ᵇ ⊢ !b ⇒ Int^∅
```

Expressions may also observe resources indirectly without referring to free variables. In the example below, the nested reference `c` allows the expression to observe not only `c`, but also `b`:

```
let c = ref b; !(!c)   // [···, c: Ref[Ref[Int]ᵇ]♦]ᵇ,ᶜ ⊢ !(!c) ⇒ Int^∅
```

In $G^{\blacklozenge}_{\Leftrightarrow}$, while typing expressions, we additionally collect their observations. Function qualifiers then include their body observations, governing the resources they may access when invoked.

```
let b = ref 42; let c = ref b
fun () => !(!c)        // [···]ᵇ,ᶜ ⊢ <fun> ⇒ (Unit → Int)ᵇ,ᶜ
```

Constraints from Subtyping. Given an existing function, subtype checking ($\leqslant$) is required when we need to convert it into a different type, as exemplified below:

$$[\cdots, \texttt{a} : \texttt{Ref[Int]}^{\blacklozenge}, \texttt{b} : \texttt{Ref[Int]}^{\blacklozenge}] \quad \vdash \quad (\texttt{Unit} \to \texttt{Ref[Int]}^{\texttt{b}})^{\texttt{a}} \quad \leqslant \quad (\texttt{f(Unit)} \to \texttt{Ref[Int]}^{\texttt{f}})^{???}$$

While the qualifier after conversion (???) needs inference, we know it should be at least `a` and satisfy typing constraints from the self-reference `f` used in the type.

In $G^{\blacklozenge}_{\Leftrightarrow}$, we take inspiration from the *eager instantiation* approach [22] for higher-ranked polymorphic type inference. We make the initial guess for ??? to be `a,□`, where the *qualifier hole* □ stands for the existential reachability to be inferred, analogous to unification variables for type inference

but occurring only in typing contexts. In the typing context for checking function subtyping, we use this initial guess to qualify the self-reference `f`:

$$[\cdots, \texttt{a} : \texttt{Ref[Int]}^{\blacklozenge}, \texttt{b} : \texttt{Ref[Int]}^{\blacklozenge}, \texttt{f} : (\cdots)^{\texttt{a,}\square}] \quad \vdash \quad \texttt{b} \quad < \quad \texttt{f}$$

During the checking procedure, the qualifier subsumption obligation `b < f` poses a constraint to the reachability of `f`. To eagerly satisfy this constraint, we insert `b` into the hole attached to `f`:

$$[\cdots, \texttt{a} : \texttt{Ref[Int]}^{\blacklozenge}, \texttt{b} : \texttt{Ref[Int]}^{\blacklozenge}, \texttt{f} : (\cdots)^{\texttt{a,b,}\square}] \quad \vdash \quad \texttt{b} \quad < \quad \texttt{f}$$

When completing the function subtype checking, the guess has been updated to $\texttt{a,b,}\square$. We then *seal* the hole and use `a,b` as the final answer to the qualifier inferred in subtype checking:

$$[\cdots, \texttt{a} : \texttt{Ref[Int]}^{\blacklozenge}, \texttt{b} : \texttt{Ref[Int]}^{\blacklozenge}] \quad \vdash \quad (\texttt{Unit} \to \texttt{Ref[Int]}^{\texttt{b}})^{\texttt{a}} \quad \leqslant \quad (\texttt{f(Unit)} \to \texttt{Ref[Int]}^{\texttt{f}})^{\texttt{a,b}}$$

2.6 Summary

Self-references in reachability types are crucial for encoding data types, whereas their presence in prior work hinders devising an avoidance conversion scheme and complicates qualifier inference (Section 2.2). In this work, we first present a refined declarative calculus $\mathrm{G}^{\blacklozenge}_{<:}$ (Section 3) that combines type-and-qualifier subtyping and enables self-references in all covariant positions for expressiveness and soundness (Sections 2.4 and 3.3). Building on this theoretical foundation, we develop the algorithmic solution $\mathrm{G}^{\blacklozenge}_{\Leftrightarrow}$ (Section 4) that automatically avoids ill-scoped variables by self-references in a polarity-guided fashion (Sections 2.3 and 4.3) and infers qualifiers for all expressions, including new function definitions and their type conversions (Sections 2.5, 4.2 and 4.4).

3 $\mathrm{G}^{\blacklozenge}_{<:}$: Declarative Typing Specification

We present the formal theory and metatheory of $\mathrm{G}^{\blacklozenge}_{<:}$, a refined variant of the polymorphic reachability type system $\mathrm{F}^{\blacklozenge}_{<:}$ [53]. To serve as a specification for our algorithmic development, $\mathrm{G}^{\blacklozenge}_{<:}$ adopts shallow, dual-component reference types [16], introduces combined type-and-qualifier subtyping (Section 3.3), and uses algorithmic contexts with qualifier holes. The type system and its soundness proofs are fully mechanized in Lean 4. In the following sections, we detail the design of $\mathrm{G}^{\blacklozenge}_{<:}$, and briefly discuss its metatheory in Section 3.4.

3.1 Syntax Definitions and Well-Formedness

Figure 1 presents the syntax of $\mathrm{G}^{\blacklozenge}_{<:}$, which is based on System $F_{<:}$ with higher-order references. In our extended version [29], we provide the well-formedness definitions for the syntax of $\mathrm{G}^{\blacklozenge}_{<:}$.

Terms. Terms include constants, variables, references and operations, function abstractions and applications, and type abstractions and applications. For clarity, we use distinct metavariables for ordinary variables (*i.e.*, x, y, z), functions (*i.e.*, f, g, h), and type variables (*i.e.*, X, Y, Z). Conventionally, ordinary variables such as x may represent functions, but not the other way around. For function abstraction $\lambda f(x : Q).\ t$, we read f as the self-reference of the function, and x as the name of the argument; the argument type Q is optional. Similarly for $\Lambda f[X^{x} <: T^{q}].\ t$, we read f as the self-reference, $X <: T$ as the type variable quantification and $x <: q$ as the qualifier quantification. We also add type ascription $(t : Q)$ to support the bidirectional typing algorithm (cf. Section 4).

Qualifiers and Types. Qualified types (Q) consist of a type (T) paired with a qualifier (q). Qualifiers are sets of variables that may include the freshness marker $\blacklozenge$, denoting fresh values without names. Types include the base type B, references, functions, type variables, the Top, and universal types. The reference types are dual-component [16]: the first Q describes the type for putting, and the second describes getting; we simply write Ref Q if the two components are the same.

Syntax $\mathrm{G}^{\blacklozenge}_{<:}$

x, y, z	$\in$	Var	Variables
f, g, h	$\in$	Var	Self Variables
X, Y, Z	$\in$	TVar	Type Variables
$\square, \square?$	$\in$	QHole	(Optional) Qualifier Holes
t	$::=$	$c \mid x \mid \mathsf{ref}\ t \mid\ !\,t \mid t := t \mid t\ t \mid t\ [Q] \mid (t : Q) \mid$ $\lambda f(x : Q^{?}).\ t \mid \Lambda f[X^{x} <: Q^{?}].\ t$	Terms (with optional domain types)
p, q, r, s	$\in$	$\mathcal{P}_{\mathsf{fin}}(\mathsf{Var} \uplus \{\blacklozenge\})$	Reachability Qualifiers
P, Q, R, S	$::=$	$T^{\,q}$	Qualified Types
T, U, V, W	$::=$	$\mathsf{B} \mid \mu h.\, \mathsf{Ref}\ Q..Q \mid f(x : Q) \to Q \mid$ $X \mid \mathsf{Top} \mid \forall f[X^{x} <: Q].\ Q$	Types
Γ	$::=$	$\varnothing \mid \Gamma, x : Q \mid \Gamma, X^{x} <: Q \mid \Gamma, f : T^{\,q,\square?}$	Typing Environments
φ	$\in$	$\mathcal{P}_{\mathsf{fin}}(\mathsf{Var})$	Observations

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

Substitution, Reachability and Overlap $\quad q[p/x] \quad \Gamma \vdash x \rightsquigarrow x \quad \Gamma \vdash q* \quad \Gamma \vdash p \mathbin{⦶} q$

Qualifier Substitution	$q[p/x] := q \backslash \{x\} \cup p$, if $x \in q$;	$q[p/x] := q$,	otherwise.
Reachability Relation	$\Gamma \vdash x \rightsquigarrow y \Leftrightarrow x : T^{q,y} \in \Gamma$	Variable Saturation	$\Gamma \vdash x* := \{\, y \mid x \rightsquigarrow^{*} y \,\}$
Qualifier Saturation	$\Gamma \vdash q* := \bigcup_{x \in q} x*$	Qualifier Overlap	$\Gamma \vdash p \mathbin{⦶} q := \blacklozenge(p* \cap q*)$

Fig. 1. Syntax definitions of $\mathrm{G}^{\blacklozenge}_{<:}$ with qualifier shorthands and qualifier operations. Sometimes the context Γ is implicit (in gray). Adapted from Wei et al. [53], $\mathrm{G}^{\blacklozenge}_{<:}$ introduces qualifier holes in typing contexts and a type ascription term, and adopts dual-component reference types [16]. We emphasize them as shaded .

References, functions, and universal types also include self-references h or f in their types, and we constrain the occurrences of such self-references. As motivated in Section 2.4 and illustrated by (C-FUN) in our extended version [29], duplicated here:

$$\frac{\Gamma, f : \mathsf{Top}^{\blacklozenge} \vdash T^{\,p} \qquad \Gamma, f : \mathsf{Top}^{\blacklozenge}, x : T^{\,p} \vdash U^{\,q} \qquad f \notin^{+} T \qquad f \notin p \qquad f \notin^{-} U}{\Gamma \vdash f(x : T^{\,p}) \to U^{\,q}} \tag{C-FUN}$$

The self-reference f must not appear in covariant positions of the domain T (*i.e.*, $f \notin^{+} T$), contravariant positions of the codomain U (*i.e.*, $f \notin^{-} U$), and the domain qualifier p. These polarity constraints are crucial to the soundness of our subtyping extension.

Algorithmic Contexts. We use the metavariable Γ to denote typing contexts. In $\mathrm{G}^{\blacklozenge}_{<:}$, these contexts are algorithmic [22]: their entries are ordered, allowing insertion and deletion in the back.

Self-reference entries in contexts may contain qualifier holes, *e.g.*, $f : T^{\,q,\square} \in \Gamma$. For other entries, qualifiers must be fully specified, free of holes. Definitions of well-formed contexts are given in our extended version [29], where all entries must be closed under their preceding contexts.

Qualifier Shorthands and Operations. For presentation purposes, we define qualifier shorthands in Figure 1, allowing notation such as x for singleton sets $\{x\}$ and p, q for their union $p \cup q$. We also define qualifier substitution and overlap, both used in application rules. Substitution on types is standard and thus omitted. Qualifier overlap is defined as the intersection of transitive reachability closures, and is used in the separation judgment (F-FRESH).

3.2 Typing

Figure 2 presents the typing rules of $\mathrm{G}^{\blacklozenge}_{<:}$, written in the form $\Gamma \overset{\varphi}{\vdash} t : Q$, where φ is the *observation filter*, summarizing all free variables required to type the term t. The typing rules mainly follow the design of prior works [16, 53], but build on the subtyping rules in Section 3.3. We use $\leqslant:$ to

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

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

$$\frac{\Gamma^{\varphi} \vdash t : T^{q} \qquad \blacklozenge \notin q}{\Gamma^{\varphi} \vdash \mathrm{ref}\ t : (\mathrm{Ref}\ T^{q})^{\blacklozenge}}\ (\textsc{t-ref}) \qquad \frac{\Gamma^{\varphi} \vdash t : (\mu h.\, \mathrm{Ref}\ P..T^{q})^{p} \qquad q \subseteq h, \varphi \qquad \blacklozenge \notin p \vee h \notin T}{\Gamma^{\varphi} \vdash\, !\, t : T^{q}[p/h]}\ (\textsc{t-get})$$

$$\frac{\Gamma^{\varphi} \vdash t_1 : (\mu h.\, \mathrm{Ref}\ P..Q)^{p} \qquad \Gamma^{\varphi} \vdash t_2 : P}{\Gamma^{\varphi} \vdash t_1 := t_2 : \mathrm{B}^{\varnothing}}\ (\textsc{t-put})$$

$$\frac{(\Gamma,\ f{:}\mathrm{Top}^{q},\ x{:}T^{p})^{q,f,x} \vdash t : Q \qquad p \subseteq \blacklozenge q \qquad q \subseteq \varphi}{\Gamma^{\varphi} \vdash \lambda f(x).\, t : (f(x{:}T^{p}) \to Q)^{q}}\ (\textsc{t-abs}) \qquad \frac{\Gamma^{\varphi} \vdash t_1 : (f(x{:}T^{p}) \to U^{r})^{q} \qquad \Gamma^{\varphi} \vdash t_2 : T^{s} \qquad \Gamma^{\varphi} \vdash s \prec:^{q} p \qquad r \subseteq \blacklozenge\varphi, f, x \qquad \blacklozenge \notin s \vee x \notin U \qquad \blacklozenge \notin q \vee f \notin U}{\Gamma^{\varphi} \vdash t_1\ t_2 : U^{r}[s/x, q/f]}\ (\textsc{t-app})$$

$$\frac{(\Gamma,\ f{:}\mathrm{Top}^{q},\ X^{x}{<:}T^{p})^{q,f,x} \vdash t : Q \qquad p \subseteq \blacklozenge q \qquad q \subseteq \varphi}{\Gamma^{\varphi} \vdash \Lambda f[X^{x}].\, t : (\forall f[X^{x}{<:}T^{p}].\, Q)^{q}}\ (\textsc{t-tabs}) \qquad \frac{\Gamma^{\varphi} \vdash t : (\forall f[X^{x}{<:}T^{p}].\, U^{r})^{q} \qquad \Gamma \vdash V <: T \qquad s \subseteq \blacklozenge\varphi \qquad \Gamma^{\varphi} \vdash s \prec:^{q} p \qquad r \subseteq \blacklozenge\varphi, f, x \qquad \blacklozenge \notin s \vee x \notin U \qquad \blacklozenge \notin q \vee f \notin U}{\Gamma^{\varphi} \vdash t\ [V^{s}] : U^{r}[V^{s}/X^{x}, q/f]}\ (\textsc{t-tapp})$$

Application Conformance $\boxed{\Gamma^{\varphi} \vdash q \prec:^{q} q}$

$$\frac{\Gamma \vdash s <: p}{\Gamma^{\varphi} \vdash s \prec:^{q} p}\ (\textsc{f-sub}) \qquad \frac{\Gamma \vdash s \pitchfork q <: \blacklozenge p \qquad s \pitchfork q \subseteq \blacklozenge\varphi \qquad \square \notin s*, q*}{\Gamma^{\varphi} \vdash s \prec:^{q} \blacklozenge p}\ (\textsc{f-fresh})$$

Fig. 2. Select typing rules of $\mathrm{G}^{\blacklozenge}_{<:}$. Adapted from Wei et al. [53], the rules are presented with explicit conformance to merge their two separate application rules, and with shallow, dual-component reference types [16]. Rules for type annotations are available in our extended version [29].

denote the new combined type-and-qualifier subtyping relations, and $<:$ for the separate relations of qualifiers and types, as appeared in prior work [53].

Basic typing rules handle constants and variables. For constants (T-CST), the empty qualifier is assigned, as primitive values do not track resources. For the variable x (T-VAR), the qualifier is x, regardless of the qualifier q recorded in the context. This q can later be revealed via subsumption; see (Q-VAR) in Figure 3. Additionally, x must appear in the observation φ.

Allocation (T-REF) yields a reference shallowly qualified by only $\blacklozenge$. Aligned with prior work [5, 53], the referent must be non-fresh. Both components in the resulting type are the same, thus abbreviated. Dereferencing (T-GET) replaces the self-reference h in the *get* component with the reference qualifier itself. Such substitution of bound variables observes the same restriction seen in Section 2.2 and has to be either non-fresh ($\blacklozenge \notin p$) or non-deep ($h \notin T$). Assigning the reference (T-PUT) concerns the *put* component and is otherwise standard.

Subsumption (T-SUB) and ascription are the foundation for mode switching in bidirectional typing. In (T-SUB), we apply extended subtyping ($\leqslant:$) to enable expressive conversions involving self-references. The qualifier q of the supertype must be bounded by the filter φ. The ascription rule and the rules for abstractions with domain annotations trivially delegate to unannotated terms and are left for our extended version [29].

The abstraction rule (T-ABS) introduces both the self-reference f and the argument variable x. Representing only reachability, the self-reference f is given the top type with the function qualifier q. This q extended with f and x then restricts the observation for the body t.

The application rule (T-APP) requires the parameter and the argument to have the same type T, but allows different qualifiers. Reflecting the two separate application rules in prior works [16, 53], the parameter qualifier s must either be bounded by p (F-SUB), or overlap with the function qualifier q by no more than p (F-FRESH). Holes are rejected in saturation computation, ensuring that future instantiations do not affect the result. Besides the two cases, we mechanize an extension [28, 29]

Subqualifying $\boxed{\Gamma \vdash q <: q}$

$$\frac{p \subseteq q}{\Gamma \vdash p <: q}\ (\text{Q-SUB}) \qquad \frac{\Gamma \vdash p <: q \qquad \Gamma \vdash q <: r}{\Gamma \vdash p <: r}\ (\text{Q-TRANS}) \qquad \frac{\Gamma \vdash p <: r \qquad \Gamma \vdash q <: s}{\Gamma \vdash p, q <: r, s}\ (\text{Q-CONG})$$

$$\frac{x : T^{\,q} \in \Gamma \qquad \blacklozenge \notin q}{\Gamma \vdash x <: q}\ (\text{Q-VAR}) \qquad \frac{X^{x} <: T^{\,q} \in \Gamma \qquad \blacklozenge \notin q}{\Gamma \vdash x <: q}\ (\text{Q-TVAR}) \qquad \frac{f : T^{\,q,\square?} \in \Gamma}{\Gamma \vdash q \backslash \blacklozenge <: f}\ (\text{Q-SELF})$$

Type-and-Qualifier Subtyping $\boxed{\Gamma \vdash Q \leqslant: Q}$

$$\frac{\Gamma \vdash p <: q}{\Gamma \vdash T^{\,p} \leqslant: T^{\,q}}\ (\text{S-GROW}) \qquad \frac{\Gamma \vdash P \leqslant: Q \qquad \Gamma \vdash Q \leqslant: R}{\Gamma \vdash P \leqslant: R}\ (\text{S-TRANS})$$

$$\frac{(\Gamma, h : \mathrm{Top}^{\,q})^{\,\varphi_1} \vdash Q \leqslant: P \qquad \varphi_1 \subseteq q, h \qquad (\Gamma, h : \mathrm{Top}^{\,q})^{\,\varphi_2} \vdash R \leqslant: S \qquad \varphi_2 \subseteq q, h}{\Gamma \vdash (\mu h.\, \mathrm{Ref}\ P..R)^{\,q} \leqslant: (\mu h.\, \mathrm{Ref}\ Q..S)^{\,q}}\ (\text{S-REF})$$

$$\frac{(\Gamma, f : \mathrm{Top}^{\,q})^{\,\varphi_1} \vdash Q \leqslant: P \qquad \varphi_1 \subseteq q, f \qquad (\Gamma, f : \mathrm{Top}^{\,q}, x : Q)^{\,\varphi_2} \vdash R[(x, \varphi_1)/x] \leqslant: S \qquad \varphi_2 \subseteq q, f, x}{\Gamma \vdash (f(x : P) \to R)^{\,q} \leqslant: (f(x : Q) \to S)^{\,q}}\ (\text{S-FUN})$$

$$\frac{}{\Gamma \vdash T^{\,q} \leqslant: \mathrm{Top}^{\,q}}\ (\text{S-TOP}) \qquad \frac{X^{x} <: T^{\,p} \in \Gamma}{\Gamma \vdash X^{\,q} \leqslant: T^{\,q}}\ (\text{S-TVAR}) \qquad \frac{(\Gamma, f : \mathrm{Top}^{\,q})^{\,\varnothing} \vdash Q \leqslant: P \qquad (\Gamma, f : \mathrm{Top}^{\,q}, X^{x} <: Q)^{\,\varphi_2} \vdash R \leqslant: S \qquad \varphi_2 \subseteq q, f, x}{\Gamma \vdash (\forall f[X^{x} <: P].\, R)^{\,q} \leqslant: (\forall f[X^{x} <: Q].\, S)^{\,q}}\ (\text{S-ALL})$$

Subtyping Shorthands $\boxed{\Gamma \vdash T <: T}$ $\boxed{\Gamma^{\,\varphi} \vdash Q \leqslant: Q}$

$$\frac{\Gamma \vdash T^{\blacklozenge} \leqslant: U^{\blacklozenge}}{\Gamma \vdash T <: U}\ (\text{S-UNCOND}) \qquad \frac{\Gamma \vdash T^{\blacklozenge} \leqslant: U^{\blacklozenge\varphi} \qquad \Gamma \vdash p <: q \qquad \Gamma \vdash \varphi <: q}{\Gamma^{\,\varphi} \vdash T^{\,p} \leqslant: U^{\,q}}\ (\text{S-PROXY})$$

Fig. 3. Subsumption rules of $\mathrm{G}^{\blacklozenge}_{<:}$. Subqualifying rules mainly follow the prior work [53], with (Q-SELF) generalized upon freshness and qualifier holes. The combined type-and-qualifier subtyping is newly proposed.

that may accept parameters with arbitrary reachability. To complete the application, we substitute f and x in the result type with q and s, respectively. Similar to the case of (T-GET), such substitutions have to be either non-fresh or non-deep, according to Section 2.2.

Type abstractions (T-TABS) and applications (T-TAPP) support bounded quantification. A bound of the form $X^{x} <: T^{\,p}$ can be read as a combined type bound $X <: T$ and qualifier bound $x <: p$, where X and x may be used independently. These rules parallel their function counterparts. In (T-TAPP), the type argument V may differ from the bound T; their subtyping is checked using the basic relation ($<:$), independent of qualifiers.

3.3 Subtyping and Subqualifying

We present subsumption rules of $\mathrm{G}^{\blacklozenge}_{<:}$ in Figure 3, including subqualifying rules that mainly follow prior design and type-and-qualifier subtyping rules introduced in Section 2.4.

Subqualifying. Qualifiers are sets and thus the subset relation is carried over as (Q-SUB). In addition, (Q-TRANS) and (Q-CONG) are defined in a similar way to the transitivity and congruence rules of subset. Other rules leverage reachability information from the typing context Γ. Rules (Q-VAR) and (Q-TVAR) expand variables by replacing them with their recorded qualifiers in Γ, provided these qualifiers are fully specified, *i.e.*, contain no holes or freshness. Rule (Q-SELF) introduces self-references to upper-bound the variables contained in the qualifier of the corresponding self-reference. Conversely, self-references can be expanded using (Q-VAR), as long as their qualifiers are fully established and non-fresh—conditions not required in (Q-SELF).

Combined Type-and-Qualifier Subtyping. Individual subtyping judgments may change types and/or qualifiers, and the transitivity rule (S-TRANS) allows composing such changes. Rule (S-GROW) allows changing qualifiers when types are the same between the subtype and the supertype; a

standard reflexivity rule can be derived, requiring both types and qualifiers to be the same. All other rules change only the types, requiring the same qualifiers on both sides.

As discussed earlier in Section 2.4, combined type-and-qualifier subtyping allows justifying avoidance coercions. For this purpose, rules (S-REF), (S-FUN) and (S-ALL) add the qualifier q agreed between the subtype and the supertype into the typing context, as the qualifier of their corresponding self-references. Together with the subqualifying rule for self-references (Q-SELF), this enables expressing the basic avoidance subtyping examples in Section 2.4.

When deriving subtyping for the component types, rules (S-REF), (S-FUN) and (S-ALL) use the auxiliary form (S-PROXY): they understand the qualifier p of T as *some reachability*, and q of U as *some reachability including* φ, where the freshness markers do not mean separation. This way, in (S-REF) and (S-FUN), we require only φ_1, φ_2 to be within q modulo bound variables, but do not restrict the qualifiers of P, Q, R, S; this is crucial to precise, shallow reference types [16], where referent qualifiers are not necessarily smaller than outer ones. In (S-FUN), when checking subtyping between the domain types and qualifiers R and S, it uses the substitution $R[(x, \varphi_1)/x]$ to account for the fact that x in R and S may refer to arguments with reachability differing by φ_1.

We further include (S-TOP), (S-TVAR), and (S-ALL) for type polymorphism. The former two rules are standard. Rule (S-ALL) behaves similarly to (S-FUN), except that the subtyping between type-and-qualifier bounds via (S-PROXY) requires no observable φ_1, so that type bounds and qualifier bounds can be used orthogonally. This aligns with type applications (T-TAPP), where we use the type-only subtyping (by <:) that applies regardless of qualifiers. In $\mathrm{G}^{\blacklozenge}_{<:}$, such type-only subtyping is not a separate set of rules, but can be derived from type-and-qualifier subtyping via (S-UNCOND), where both sides agree on the opaque qualifier $\blacklozenge$. This is also how subtyping rules from prior work [16, 53] can be understood in the context of this work.

3.4 Metatheory

3.4.1 Semantic Soundness. Unlike prior work [6, 16, 53] that establishes syntactic soundness, we prove the type soundness of $\mathrm{G}^{\blacklozenge}_{<:}$ using *logical relations* [50]. The dynamic semantics is formulated as a big-step interpreter [3]. Our semantic interpretation is adapted from Bao et al. [5], and we extend their model by adding interpretations for type polymorphism, shallowly qualified, dual-component references [16], and deep occurrences of bound variables inside types. Besides efforts to reflect recent advances in reachability types, our new subtyping design requires our semantic model to interpret the reachable locations of values in a type-dependent manner, and this shift necessitated nontrivial changes. Details of our logical relations are provided in our extended version [29], and we excerpt the key results as follows.

THEOREM 3.1 (FUNDAMENTAL). *If a term t is syntactically well-typed,* i.e., $\Gamma^{\varphi} \vdash t : T^{q}$, *and both the context* Γ *and the store* σ *are well-formed, then* t *is also semantically well-typed* ($\Gamma \models t : T^{q}$). *Specifically,* t *evaluates to a value* v *of the type* T *in finite steps, such that* v *may only reach locations described by the qualifier* q, *and all store write effects are limited to the locations described by* φ.

In particular, for terms closed under the empty context, we obtain a formulation of type safety that requires no assumption about contexts or stores: *well-typed terms do not get stuck.*

COROLLARY 3.2 (TYPE SAFETY). *If* $\varnothing \vdash t : T^{\varnothing}$, *then* t *evaluates to a value* v *of type* T *in finite steps.*

The *Preservation of Separation* property [53] also follows from our analysis of store effects: the evaluation of two well-typed terms with disjoint observations will observe and update disjoint portions of the store.

3.4.2 Interacting with Qualifier Holes. Although $\mathrm{G}^{\blacklozenge}_{<:}$ includes no rules that introduce qualifier holes, we show that inserting and instantiating such holes preserves the soundness results. We

Context Subsumption $\boxed{\Gamma \sqsubseteq \Gamma}$

$$\frac{}{\varnothing \sqsubseteq \varnothing} \qquad \frac{\Gamma \sqsubseteq \Gamma'}{\Gamma, x{:}Q \sqsubseteq \Gamma', x{:}Q} \qquad \frac{\Gamma \sqsubseteq \Gamma'}{\Gamma, X^x{<:}Q \sqsubseteq \Gamma', X^x{<:}Q} \qquad \frac{\Gamma \sqsubseteq \Gamma' \qquad \Gamma \vdash q \qquad \blacklozenge \notin q}{\Gamma, f{:}T^{p,\square} \sqsubseteq \Gamma', f{:}T^{p,q,\square}}$$

Fig. 4. Definitions of context subsumption: context entries are either the same, or with their qualifier holes instantiated by well-formed, non-fresh qualifiers.

define *context subsumption* in Figure 4 to characterize the effect of hole instantiation. The relation $\Gamma \sqsubseteq \Gamma'$ indicates that Γ' results from partially instantiating holes in Γ zero or more times.

Lemma 3.3 (Context Subsumption on Typing). *If* $\Gamma\,^{\varphi} \vdash t : Q$, *and* $\Gamma \sqsubseteq \Gamma'$, *then* $\Gamma'\,^{\varphi} \vdash t : Q$.

Lemma 3.4 (Hole Sealing on Typing). *If* $(\Gamma_1, f{:}T^{q,\square}, \Gamma_2)\,^{\varphi} \vdash t : Q$, *then* $(\Gamma_1, f{:}T^{q}, \Gamma_2)\,^{\varphi} \vdash t : Q$.

These lemmas show that proving a type judgment for a specific context Γ can be reduced to proving it under a *weaker* context–one containing more holes or instantiations with smaller qualifiers. This property is crucial for the soundness of qualifier inference in the algorithm.

4 $G^{\blacklozenge}_{\Leftrightarrow}$: Bidirectional Typing with Qualifier Inference and Avoidance

The prior presentation of $G^{\blacklozenge}_{<:}$ is *declarative*, not specifying which qualifiers can be inferred and by what means. In this section, we introduce its typing algorithm, $G^{\blacklozenge}_{\Leftrightarrow}$. Following bidirectional typing [21], $G^{\blacklozenge}_{\Leftrightarrow}$ infers both types and qualifiers for terms, given annotations on function arguments and explicit instantiations of type abstractions. When a fresh value escapes within other resources, the algorithm automatically applies avoidance conversions to track the freshness via self-references. We have implemented $G^{\blacklozenge}_{\Leftrightarrow}$ in Lean and proven its soundness with respect to $G^{\blacklozenge}_{<:}$. We also prove its termination, at the cost of being incomplete and rejecting some valid $G^{\blacklozenge}_{<:}$ terms.

We organize the algorithm presentation bottom-up, covering qualifiers (Section 4.1), subtyping (Section 4.2), avoidance (Section 4.3), and finally bidirectional typing (Section 4.4). We briefly discuss the metatheoretical properties of $G^{\blacklozenge}_{\Leftrightarrow}$ in Section 4.5.

4.1 Qualifier Checking and Inference

Figure 5 presents the rules for qualifier checking and inference in $G^{\blacklozenge}_{\Leftrightarrow}$, corresponding to the subqualifying rules in Figure 3. These rules can be divided into two main components: *qualifier exposure* without considering qualifier holes, and *unification* that instantiates holes.

Qualifier Exposure and Checking. Qualifier exposure ($\Uparrow$) for q aims to find a large enough qualifier q', so that checking $p <: q$ can be reduced to simply checking $p \subseteq q'$, as seen in (QCHECK).

The exposure procedure (QE-JOIN) proceeds in two stages. In both stages, we extend the input qualifier by a subqualifier. The first stage $\Uparrow_1$ (QE-SELF) enumerates self-references in the input qualifier, and extends the input qualifier with the qualifiers of the found self-references, reflecting the declarative rule (Q-SELF). The second stage $\Uparrow_2$ (QE-VAR) then adds all variables whose reachability is already included in the input qualifier, reflecting the declarative rule (Q-VAR).

Although applied iteratively, neither step diverges. For well-formed contexts, the first stage can finish by scanning the context once in the reverse order, while the second stage can finish by scanning once in the forward order.

Qualifier Unification and Inference. Qualifier checking and inference differ in their approach to constraint satisfaction. While qualifier checking ignores all qualifier holes, qualifier inference (QINFER) replaces subset checking $\subseteq$ with qualifier unification $\subseteq?$, eagerly satisfying constraints by instantiating qualifier holes in the context.

Qualifier Exposure $\boxed{\Gamma \vdash q \Uparrow q}$ $\boxed{\Gamma \vdash q \Uparrow_1 q}$ $\boxed{\Gamma \vdash q \Uparrow_2 q}$

$$\frac{\Gamma \vdash q \Uparrow_1{}^* q_1 \qquad \Gamma \vdash q_1 \Uparrow_2{}^* q_2}{\Gamma \vdash q \Uparrow q_2} \text{ (QE-JOIN)}$$

$$\frac{f{:}T^{p,\square?} \in \Gamma \qquad p \backslash \blacklozenge \not\subseteq q}{\Gamma \vdash f, q \;\Uparrow_1\; f, q, (p \backslash \blacklozenge)} \text{ (QE-SELF)} \qquad \frac{x{:}T^{p} \in \Gamma \vee X^{x}{<:}T^{p} \in \Gamma \qquad \blacklozenge \notin p \qquad x \notin q}{\Gamma \vdash p, q \;\Uparrow_2\; p, q, x} \text{ (QE-VAR)}$$

Qualifier Unification $\boxed{\Gamma \vdash q \subseteq? \, q \dashv \Gamma}$

$$\frac{x \notin p, q \qquad \begin{array}{l}\Gamma = .., x{:}Q, \quad .., f{:}T^{r,\square}, .. \\ \vee\, \Gamma = .., X^{x}{<:}Q, .., f{:}T^{r,\square}, ..\end{array} \qquad \Gamma \vdash p \subseteq? \, q, f \dashv \Gamma_1, f{:}T^{s,\square}, \Gamma_2}{\Gamma \vdash p, x \subseteq? \, q, f \dashv \Gamma_1, f{:}T^{s,x,\square}, \Gamma_2} \text{ (QU}_1\text{-UNIFY)}$$

$$\frac{x \notin p, q \qquad x{:}T^{r} \in \Gamma \vee X^{x}{<:}T^{r} \in \Gamma \qquad \blacklozenge \notin r \qquad \Gamma \vdash p, r \subseteq? \, q \dashv \Gamma'}{\Gamma \vdash p, x \subseteq? \, q \dashv \Gamma'} \text{ (QU}_2\text{-UPCAST)} \qquad \frac{p \subseteq q}{\Gamma \vdash p \subseteq? \, q \dashv \Gamma} \text{ (QU}_3\text{-BOT)}$$

Qualifier Checking and Inference $\boxed{\Gamma \vdash q < q}$ $\boxed{\Gamma \vdash q < q \dashv \Gamma}$

$$\frac{\Gamma \vdash q \Uparrow q' \qquad p \subseteq q'}{\Gamma \vdash p < q} \text{ (QCHECK)} \qquad \frac{\Gamma \vdash q \Uparrow q' \qquad \Gamma \vdash p \subseteq? \, q' \dashv \Gamma'}{\Gamma \vdash p < q \dashv \Gamma'} \text{ (QINFER)}$$

Fig. 5. Qualifier checking and inference in $G^{\blacklozenge}_{\Leftrightarrow}$. In the order of the indices, rules of the same form are tried sequentially, and the first succeeding one is applied. Outputs in the rules are marked in purple.

Qualifier unification concludes by (QU$_3$-BOT) when p is simply a subset of q. Whenever this is not the case, there must be an outstanding variable x that does not appear in q. Primarily, the unification rule (QU$_1$-UNIFY) tries to insert x into the qualifier hole of a self-reference f. This unification step is the key to satisfying `b < f` in the example from Section 2.5.

Crucially, the choice of f is not arbitrary: (1) f must be defined after x, so that the instantiation yields a well-formed context, and (2) among all candidates, we select the one defined earliest to avoid cascading updates on others. If no such f exists, we apply (QU$_2$-UPCAST) to replace x with its reachability r. This is allowed only if r contains no holes or freshness; otherwise, unification fails.

Unification processes all relevant context entries in reverse order, each exactly once. To infer qualifiers in subtyping and typing as illustrated in Section 2.5, we use (QINFER) but not (QCHECK).

4.2 Subtype Checking

Figure 6 presents the rules for subtype checking, which also infers the qualifier for the supertype. The top-level procedure (SA-JOIN) operates in two phases: *self unpacking* and *recursive checking*. We write $\leqslant$ for the algorithm, in contrast to the declarative $\leqslant:$.

Recursive Subtype Checking. The second phase $\leqslant_2$ adapts the declarative subtyping rules. Here, (S-GROW) is specialized into (SA-BASE) and (SA-TVAR$_1$), while (S-TRANS) is internalized in (SA-TVAR$_2$). Rule (SA-ALL) implements a kernel variant of $F_{<:}$ for decidability, unlike the declarative (S-ALL) based on the full variant [25]. Rules (SA-REF), (SA-FUN) and (SA-ALL) initialize the qualifier of their self-references using the input qualifier q with a hole $\square$. With (QINFER), they eagerly satisfy constraints on the hole (Section 2.5) and thus infer the additional reachability q'' to appear in the supertype qualifier. This phase alone enables the *packing* conversion illustrated in Section 2.4.

Self Unpacking. With qualifier holes in self-reference qualifiers, the recursive phase does not support the *unpacking* conversion from Section 2.4, as (Q-VAR) requires the fully specified qualifiers. To address this, we add a dedicated unpacking step prior to recursive checking, denoted $\leqslant_1$, to replace f with the input qualifier if it is not fresh. The two steps connect by transitivity (S-TRANS).

Toplevel Subtype Checking $\boxed{\Gamma \vdash Q \leqslant T^{\,q} \dashv \Gamma}$

$$\frac{T_1 \leqslant_1^p T_1' \qquad \Gamma \vdash T_1' \leqslant_2^p T_2 \nearrow^q \dashv \Gamma'}{\Gamma \vdash T_1{}^{p} \leqslant T_2{}^{p,q} \dashv \Gamma'} \text{(SA-JOIN)}$$

Self Unpacking (Selection) $\boxed{T \leqslant_1^q T}$

$$\frac{\blacklozenge \notin q \qquad \theta = [q/h]}{\mu h.\,\mathsf{Ref}\; T^{\,p}..Q \leqslant_1^q \mu h.\,\mathsf{Ref}\; T\theta^{\,p}..Q\theta} \text{(SU}_1\text{-REF)} \qquad \frac{\blacklozenge \notin q \qquad \theta = [q/f]}{f(x{:}T^{\,p}) \to Q \leqslant_1^q f(x{:}T\theta^{\,p}) \to Q\theta} \text{(SU}_2\text{-FUN)} \qquad \frac{}{T \leqslant_1^q T} \text{(SU}_4\text{-BOT)}$$

Recursive Subtype Checking $\boxed{\Gamma \vdash T \leqslant_2^q T \nearrow^q \dashv \Gamma}$

$$\frac{}{\Gamma \vdash B \leqslant_2^q B \nearrow^{\varnothing} \dashv \Gamma} \text{(SA-BASE)} \qquad \frac{\begin{array}{c}\Gamma, h{:}\,\mathsf{Top}^{\,q,\square} \vdash T_2 \leqslant_2^{\blacklozenge} T_1 \nearrow^{\varphi_1} \dashv \Gamma_1 \qquad \Gamma_1 \vdash p_2, \varphi_1 < p_1 \dashv \Gamma_2 \\ \Gamma_2 \vdash U_1 \leqslant_2^{\blacklozenge} U_2 \nearrow^{\varphi_2} \dashv \Gamma_3 \qquad \Gamma_3 \vdash r_1, \varphi_2 < r_2 \dashv \Gamma', h{:}\,\mathsf{Top}^{\,q',\square} \\ q'' = (q' \setminus q),\ (\varphi_1 \setminus h),\ (\varphi_2 \setminus h)\end{array}}{\Gamma \vdash \mu h.\,\mathsf{Ref}\; T_1{}^{p_1}..U_1{}^{r_1} \leqslant_2^q \mu h.\,\mathsf{Ref}\; T_2{}^{p_2}..U_2{}^{r_2} \nearrow^{q''} \dashv \Gamma'} \text{(SA-REF)}$$

$$\frac{}{\Gamma \vdash T \leqslant_2^q \mathsf{Top} \nearrow^{\varnothing} \dashv \Gamma} \text{(SA-TOP)} \qquad \frac{}{\Gamma \vdash X \leqslant_2^q X \nearrow^{\varnothing} \dashv \Gamma} \text{(SA-TVAR}_1\text{)} \qquad \frac{X^x{<:}U^{\,p} \in \Gamma \qquad \Gamma \vdash U \leqslant_2^q T \nearrow^{q'} \dashv \Gamma'}{\Gamma \vdash X \leqslant_2^q T \nearrow^{q'} \dashv \Gamma'} \text{(SA-TVAR}_2\text{)}$$

$$\frac{\begin{array}{c}\Gamma, f{:}\,\mathsf{Top}^{\,q,\square} \vdash T_2 \leqslant_2^{\blacklozenge} T_1 \nearrow^{\varphi_1} \dashv \Gamma_1 \qquad \Gamma_1 \vdash p_2, \varphi_1 < p_1 \dashv \Gamma_2 \qquad \theta = [(x, \varphi_1)/x] \\ \Gamma_2, x{:}\,T_2{}^{p_2} \vdash U_1\theta \leqslant_2^{\blacklozenge} U_2 \nearrow^{\varphi_2} \dashv \Gamma_3 \qquad \Gamma_3 \vdash r_1\theta, \varphi_2 < r_2 \dashv \Gamma', f{:}\,\mathsf{Top}^{\,q',\square}, .. \\ q'' = (q' \setminus q),\ (\varphi_1 \setminus f),\ (\varphi_2 \setminus \{f, x\})\end{array}}{\Gamma \vdash f(x{:}\,T_1{}^{p_1}) \to U_1{}^{r_1} \leqslant_2^q f(x{:}\,T_2{}^{p_2}) \to U_2{}^{r_2} \nearrow^{q''} \dashv \Gamma'} \text{(SA-FUN)}$$

$$\frac{\begin{array}{c}\Gamma, f{:}\,\mathsf{Top}^{\,q,\square} \vdash p_2 < p_1 \dashv \Gamma_2 \\ \Gamma_2, X^x{<:}T^{\,p_2} \vdash U_1 \leqslant_2^{\blacklozenge} U_2 \nearrow^{\varphi_2} \dashv \Gamma_3 \qquad \Gamma_3 \vdash r_1, \varphi_2 < r_2 \dashv \Gamma', f{:}\,\mathsf{Top}^{\,q',\square}, \cdots \\ q'' = (q' \setminus q),\ (\varphi_2 \setminus \{f, x\})\end{array}}{\Gamma \vdash \forall f[X^x{<:}T^{\,p_1}].\, U_1{}^{r_1} \leqslant_2^q \forall f[X^x{<:}T^{\,p_2}].\, U_2{}^{r_2} \nearrow^{q''} \dashv \Gamma'} \text{(SA-ALL)}$$

Fig. 6. Select subtype checking in $\mathrm{G}^{\blacklozenge}_{\Leftrightarrow}$. Additional rules are available in our extended version [29]. Rules are syntax-directed or ordered by indices, with outputs marked in purple.

Avoidance Core $\boxed{Q \ll_x Q}$

$$\frac{\begin{array}{c}T_1 := T[h/^{-}z] \quad p_1 = p \setminus z \\ U_1 := U[h/^{+}z] \quad r_1 = r[h/z] \\ Q = (\mu h.\,\mathsf{Ref}\; T_1{}^{p_1}..U_1{}^{r_1})^{\,q,z}\end{array}}{(\mu h.\,\mathsf{Ref}\; T^{\,p}..U^{\,r})^{\,q} \ll_z Q} \text{(AV-REF)} \qquad \frac{\begin{array}{c}T_1 := T[f/^{-}z] \quad p_1 = p \setminus z \\ U_1 := U[f/^{+}z] \quad r_1 = r[f/z] \\ Q = (f(x{:}\,T_1{}^{p_1}) \to U_1{}^{r_1})^{\,q,z}\end{array}}{(f(x{:}\,T^{\,p}) \to U^{\,r})^{\,q} \ll_z Q} \text{(AV-FUN)} \qquad \frac{\begin{array}{c}T_1 := T[f/^{-}z] \quad p_1 = p \setminus z \\ U_1 := U[f/^{+}z] \quad r_1 = r[f/z] \\ Q = (\forall f[X^x{<:}T_1{}^{p_1}].\, U_1{}^{r_1})^{\,q,z}\end{array}}{(\forall f[X^x{<:}T^{\,p}].\, U^{\,r})^{\,q} \ll_z Q} \text{(AV-ALL)}$$

Conditional Avoidance $\boxed{Q \ll_{q/x} Q}$ $\boxed{Q \ll_{q/x} \ll_{q/x} Q}$

$$\frac{\blacklozenge \notin q \vee x \notin T}{T^{\,p} \ll_{q/x} T^{\,p}} \text{(AC}_1\text{-SKIP)} \qquad \frac{Q \ll_x Q'}{Q \ll_{q/x} Q'} \text{(AC}_2\text{-AVOID)} \qquad \frac{Q \ll_{p/x} Q' \qquad Q' \ll_{q/y} Q''}{Q \ll_{p/x} \ll_{q/y} Q''} \text{(AC-DOUBLE)}$$

Fig. 7. Avoidance rules in $\mathrm{G}^{\blacklozenge}_{\Leftrightarrow}$. Rules are syntax-directed or ordered by indices, with outputs marked in purple.

4.3 Avoidance Conversion

In $\mathrm{G}^{\blacklozenge}_{\Leftrightarrow}$, we implement avoidance conversions as described in Section 2.3. Shown in Figure 7, the core form (AV-$*$) is written as $P \ll_z Q$, meaning that avoiding variable z deep inside the type P yields type Q. It is defined by *polarized substitution* [29]: using the outermost self-reference f in P to replace z in covariant positions, and removing it from contravariant ones. Since the outermost function subsumes the scope of inner ones, this choice eliminates the need to handle inner self-references and results in smaller types by subtyping.

Figure 7 also defines *conditional avoidance* (AC-$*$), invoking avoidance core only on restricted substitutions. Note that we do not define avoidance core for simple types: there, the occurrence requirement $x \notin T$ in (AC$_1$-SKIP) is trivially true, and thus they are never required for (AC$_2$-AVOID).

Bidirectional Typing (Selection) $\boxed{\Gamma^{\varphi} \vdash t \Rightarrow [\Uparrow] Q \dashv \Gamma}$ $\boxed{\Gamma^{\varphi} \vdash t \Leftarrow T^{\Rightarrow q} \dashv \Gamma}$ $\boxed{\Gamma^{\varphi} \vdash t \Leftarrow Q \dashv \Gamma}$

$$\frac{c \in \mathrm{B}}{\Gamma^{\varnothing} \vdash c \Rightarrow \mathrm{B}^{\varnothing} \dashv \Gamma} \ (\mathrm{TI}_1\text{-CST}) \qquad \frac{x{:}T^{q} \in \Gamma}{\Gamma^{x} \vdash x \Rightarrow T^{x} \dashv \Gamma} \ (\mathrm{TI}_2\text{-VAR}) \qquad \frac{\Gamma^{\varphi} \vdash t \Rightarrow T^{q} \dashv \Gamma' \quad \blacklozenge \notin q}{\Gamma^{\varphi} \vdash \mathrm{ref}\ t \Rightarrow (\mathrm{Ref}\ T^{q})^{\blacklozenge} \dashv \Gamma'} \ (\mathrm{TI}_3\text{-REF})$$

$$\frac{\Gamma^{\varphi} \vdash t \Rightarrow\Uparrow (\mu h.\, \mathrm{Ref}\ P..Q)^{r} \dashv \Gamma' \quad Q \ll_{r/h} U^{q} \quad \varphi' = \varphi, (q \backslash h)}{\Gamma^{\varphi'} \vdash\, !\, t \Rightarrow U^{q}[r/h] \dashv \Gamma'} \ (\mathrm{TI}_4\text{-GET})$$

$$\frac{\Gamma^{\varphi_1} \vdash t \Rightarrow\Uparrow (\mu h.\, \mathrm{Ref}\ T^{p}..Q)^{r} \dashv \Gamma_1 \quad \blacklozenge \notin r \vee h \notin T \quad \Gamma_1^{\varphi_2} \vdash t_2 \Leftarrow T^{p}[r/h] \dashv \Gamma_2}{\Gamma^{\varphi_1,\varphi_2} \vdash t_1 := t_2 \Rightarrow \mathrm{B}^{\varnothing} \dashv \Gamma_2} \ (\mathrm{TI}_5\text{-PUT})$$

$$\frac{(\Gamma, f{:}\mathrm{Top}^{\square}, x{:}T^{p})^{\varphi} \vdash t \Leftarrow U^{q} \dashv \Gamma', f{:}\mathrm{Top}^{q,\square}, .. \quad r = (p,q,\varphi) \setminus \{\blacklozenge f, x\}}{\Gamma^{r} \vdash \lambda f(x).\, t \Leftarrow (f(x{:}T^{p}) \to U^{q})^{\Rightarrow r} \dashv \Gamma'} \ (\mathrm{TC}_1\text{-ABS})$$

$$\frac{\Gamma, f{:}\mathrm{Top}^{\blacklozenge} \vdash T^{p} \quad f \notin^{+} T \quad f \notin p \quad (\Gamma, f{:}\mathrm{Top}^{\square}, x{:}T^{p})^{\varphi} \vdash t \Rightarrow U^{q} \dashv \Gamma', f{:}\mathrm{Top}^{q,\square}, .. \quad V := U[f/^{+}f] \quad r = (p,q,\varphi) \setminus \{\blacklozenge f, x\}}{\Gamma^{r} \vdash \lambda f(x{:}T^{p}).\, t \Rightarrow (f(x{:}T^{p}) \to V^{q})^{r} \dashv \Gamma'} \ (\mathrm{TI}_6\text{-ABS})$$

$$\frac{\Gamma^{\varphi_2} \vdash t_2 \Rightarrow T^{p} \dashv \Gamma_1 \quad \Gamma_1^{\varphi_1} \vdash \lambda f(x{:}T^{p}).\, t_1 \Rightarrow (f(x{:}T^{p}) \to Q)^{q} \dashv \Gamma_2 \quad Q \ll_{p/x} \ll_{q/f} U^{r} \quad \varphi = \varphi_1, \varphi_2, (r \backslash \{\blacklozenge f, x\})}{\Gamma^{\varphi} \vdash (\lambda f(x).\, t_1)\ t_2 \Rightarrow U^{r}[p/x, q/f] \dashv \Gamma_2} \ (\mathrm{TI}_8\text{-LET})$$

$$\frac{\Gamma^{\varphi_1} \vdash t_1 \Rightarrow\Uparrow (f(x{:}T^{p}) \to Q)^{q} \dashv \Gamma_1 \quad \blacklozenge \notin q \vee f \notin T \quad \Gamma_1^{\varphi_2} \vdash t_2 \Leftarrow T[q/f]^{\Rightarrow s} \dashv \Gamma_2 \quad \Gamma_2^{\varphi_3} \vdash s \prec^{q} p \dashv \Gamma_3 \quad Q \ll_{s/x} \ll_{q/f} U^{r} \quad \varphi = \varphi_1, \varphi_2, \varphi_3, (r \backslash \{\blacklozenge f, x\})}{\Gamma^{\varphi} \vdash t_1\ t_2 \Rightarrow U^{r}[s/x, q/f] \dashv \Gamma_3} \ (\mathrm{TI}_9\text{-APP})$$

$$\frac{\Gamma \vdash Q \quad \Gamma^{\varphi} \vdash t \Leftarrow Q \dashv \Gamma'}{\Gamma^{\varphi} \vdash (t{:}Q) \Rightarrow Q \dashv \Gamma'} \ (\mathrm{TI}_{11}\text{-AS}) \qquad \frac{\Gamma^{\varphi} \vdash t \Rightarrow T^{q} \dashv \Gamma' \quad \Gamma' \vdash T \Uparrow U}{\Gamma^{\varphi} \vdash t \Rightarrow\Uparrow U^{q} \dashv \Gamma'} \ (\mathrm{TI\text{-}EXP})$$

$$\frac{\Gamma^{\varphi} \vdash t \Rightarrow Q \dashv \Gamma_1 \quad \Gamma_1 \vdash Q \leqslant T^{q} \dashv \Gamma_2}{\Gamma^{\varphi, q\blacklozenge} \vdash t \Leftarrow T^{\Rightarrow q} \dashv \Gamma_2} \ (\mathrm{TC}_3\text{-SUB}) \qquad \frac{\Gamma^{\varphi} \vdash t \Leftarrow T^{\Rightarrow q'} \dashv \Gamma_1 \quad \Gamma_1 \vdash q' < q \dashv \Gamma_2}{\Gamma^{\varphi, q\blacklozenge} \vdash t \Leftarrow T^{q} \dashv \Gamma_2} \ (\mathrm{TQ\text{-}SUB})$$

Type Exposure $\boxed{\Gamma \vdash T \Uparrow T}$

$$\frac{X^{x}{<:}T^{q} \in \Gamma \quad \Gamma \vdash T \Uparrow U}{\Gamma \vdash X \Uparrow U} \ (\mathrm{TU}_1\text{-TVAR}) \qquad \frac{}{\Gamma \vdash T \Uparrow T} \ (\mathrm{TU}_2\text{-BOT})$$

Fig. 8. Select bidirectional typing rules in $\mathrm{G}^{\blacklozenge}_{\Leftrightarrow}$. Rules in the same form are ordered by indices, and outputs are marked in purple. Additional rules are available in our extended version [29].

4.4 Bidirectional Typing

We present typing rules for $\mathrm{G}^{\blacklozenge}_{\Leftrightarrow}$ in Figure 8, bidirectionalized [21] from Figure 2. It involves inferring ($\Rightarrow$) both type and qualifier, checking the type while inferring a qualifier ($\Leftarrow^{\Rightarrow}$), and checking ($\Leftarrow$) both type and qualifier. As described in Section 2.5, all three modes synthesize filters φ and produce output contexts to propagate partially inferred qualifiers. Mode switching is handled by ascription (TI_{11}-AS) and subsumption (TC_3-SUB) (TQ-SUB).

The declarative rules (T-CST) (T-VAR) (T-REF) become inference rules (TI_1-CST) (TI_2-VAR) (TI_3-REF). Reference operations are adapted as rules (TI_4-GET) (TI_5-PUT), relying on *type exposure* ($\Uparrow$) to upcast type variables when necessary, also defined in Figure 8. Both rules require removing self-references in their referent types by substitution, and (TI_4-GET) thus further involves avoidance.

For unannotated functions, their types must be checked (TC_1-ABS), while inference requires annotations (TI_6-ABS). In both cases, function qualifiers are inferred by collecting body observations and satisfying subtyping constraints via qualifier holes (Section 2.5). (TI_6-ABS) further removes contravariant self-references in the inferred type according to (C-FUN) by polarized substitution $[f/^{+}f]$. The rules for type abstractions are similar and left for our extended version [29].

Function application results are always inferred. The let-binding rule (TI_8-LET) infers the argument type first, then infers the function type accordingly. In contrast, the standard application rule (TI_9-APP) infers the function type first, then checks the argument type while inferring a qualifier. It additionally requires type exposure, unpacking self-references in the codomain, and checking qualifier conformance. Both rules finish inference with avoidance and qualifier substitution. The rule for type application mirrors (TI_9-APP) and is left for our extended version [29].

4.5 Metatheory

For brevity, we leave the metatheory of $G^{\blacklozenge}_{\Leftrightarrow}$ to our extended version [29]. Here, we summarize the key results: all procedures of $G^{\blacklozenge}_{\Leftrightarrow}$ terminate, and they are sound with respect to the declarative $G^{\blacklozenge}_{<:}$:

Theorem 4.1 (Decidability and Soundness of Bidirectional Typing). *If* Γ ok, *then it can be decided in finitely many steps whether there exist such* Γ', φ, T, q *that* $\Gamma^{\varphi} \vdash t \Rightarrow T^{q} \dashv \Gamma'$, *and if so,* $\Gamma \sqsubseteq \Gamma'$, $\Gamma \vdash \varphi$, $\blacklozenge \notin \varphi$, *and* $\Gamma'^{\varphi} \vdash t : T^{q}$.

By necessity, $G^{\blacklozenge}_{\Leftrightarrow}$ is incomplete. Since we implement the kernel variant of $F_{<:}$, we cannot accept terms valid only in the full variant. Besides, in our extended version [29], we show that qualifier unification and thus inference have no principal solution. Still, $G^{\blacklozenge}_{\Leftrightarrow}$ produces reasonable results [29]: qualifier exposure and thus (qcheck) are proved complete, self unpacking ($\leqslant_1$) results are proved minimal, and the avoidance strategy using the outermost self-reference is better than other choices. Given the nontrivial nature of sharing and separation reasoning, as a type-based approach, we prefer termination over completeness. In Section 5, we show that $G^{\blacklozenge}_{\Leftrightarrow}$ is expressive enough to check meaningful programs.

5 Evaluation and Discussion

In this section, we evaluate $G^{\blacklozenge}_{\Leftrightarrow}$ using programming examples involving data structures. Specifically, we show that $G^{\blacklozenge}_{\Leftrightarrow}$ is more ergonomic than prior work. Additionally, we discuss how to extend $G^{\blacklozenge}_{\Leftrightarrow}$ with implicit type instantiation, and compare it with alternative avoidance solutions.

5.1 Encoding and Natively Supporting Data Structures

Using the essential constructs of lambda calculus, $G^{\blacklozenge}_{\Leftrightarrow}$ can support data structures via Church-encodings. In addition, we have mechanized support for pairs and lists as examples of native data types. Church encodings let us experiment with different reachability patterns and validate the expressiveness of our avoidance and inference mechanisms. Once validated, we implement the same typing behavior as primitive rules, enabling ergonomic interfaces without extensive type annotations. In this section, we discuss the data structures we support natively or via encodings.

Escaping Pairs. Prior work [53] proposed two encoding styles for pairs: *transparent*, tracking fields as separate components, or *opaque*, without differentiating them. Illustrated below on the left, the pair is kept transparent within the scope of `a,b`, but has to be made opaque when escaping. Prior work requires a term-level coercion to reconstruct the pair as `OPair`. After escaping, opaque pairs have to use different eliminators (`fstO` instead of `fstT`) to access the fields.

```
// prior work: encode Pair,OPair,fstT,fstO,···
val opaque = {
  val a = new Ref(1); val b = new Ref(2)
  val transparent = Pair[···](a, b)
  fstT[···](transparent)  // specialized fstT
  OPair[···](      // require η-expansion
    fstT[···](transparent),
    sndT[···](transparent))
}                  //: μp.OPair[..ᵖ,..ᵖ]♦
fstO[···](opaque)  // specialized fstO
```

```
// this work: (natively supported)
let opaque = {
  let a = ref 1; let b = ref 2
  let transparent = Pair(a, b) //: Pair[..ᵃ,..ᵇ]
  fst(transparent)
  transparent  // automated, seamless avoidance

}              //: μp.Pair[..ᵖ,..ᵖ]♦
fst(opaque)    // same eliminator after avoidance
```

In contrast, $G^{\blacklozenge}_{\Leftrightarrow}$ (on the right) enables seamless avoidance, both via encodings and natively, that *packs* the reachability of `a,b` into the pair. After avoidance, the `opaque` pair can use the same eliminators as `transparent` pairs, without requiring programmers to duplicate eliminators and differentiate their usages. Moreover, with native support, instantiation annotations (`[···]`) are not required.

Lists with Abstract Reachability. As an extension beyond the formal development of prior work, we include lists as an example of inductive data types. Our native list type, `List[T]`, is polymorphic over the element type `T`; each element is understood to reach the same resources as the list itself. This interface mirrors Church encodings where element reachability is tracked using the list's self-reference `h`. We provide a generic `fold` operator for implementing other common list operations:

```
// alternative interface: let sum = fun [q](l: List[Ref[Int]^q]) => ···
let sum = fun (l: μh.List[Ref[Int]^h]) = l.fold(0) { i, a => !i + a }
let lst = {
  let a = ref 42; let b = ref 42; let c = ref 42
  sum(a :: nil)                      // List[Ref[Int]^a]^a        <:: μh.List[Ref[Int]^h]^a
  sum(a :: b :: nil)                 // List[Ref[Int]^{a,b}]^{a,b}  <:: μh.List[Ref[Int]^h]^{a,b}
  sum(a :: b :: c :: nil)            // List[Ref[Int]^{a,b,c}]^{a,b,c}  <:: μh.List[Ref[Int]^h]^{a,b,c}
}; sum(lst)                          // List[Ref[Int]^lst]^lst    <:: μh.List[Ref[Int]^h]^lst
```

Lists with Distinct Elements. As a showcase of the flexibility of our approach, in $G^{\blacklozenge}_{\Leftrightarrow}$, we can also encode `MList`, where all elements are guaranteed to be separate [39, 46]. The cons constructor (`::`, omitting instantiations) enforces head-tail separation. This separation enables optimizations like iteration reordering in functions such as `miter`.

```
let a = ref 42; let b = ref 42; let c = ref 42
let lst = a :: b :: c :: mnil          // context: [···, lst: MList[Ref[Int]]^{a,b,c}]
miter(lst)( fun i => i := !i + 1 )     // can be safely reordered
let lstErr = a :: b :: a :: mnil       // Error: a is not separate from b::a::mnil
```

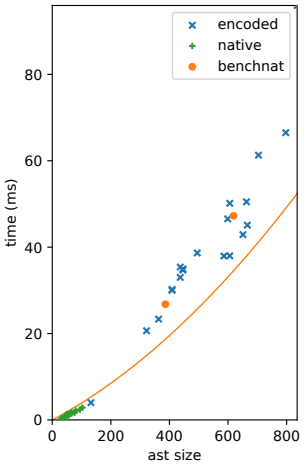
5.2 Ergonomics and Performance

To validate the ergonomics and performance of $G^{\blacklozenge}_{\Leftrightarrow}$, we implement the motivating examples from Section 1 and variations, using both Church encodings and native data types. We refer readers to our artifact [28] for full details.

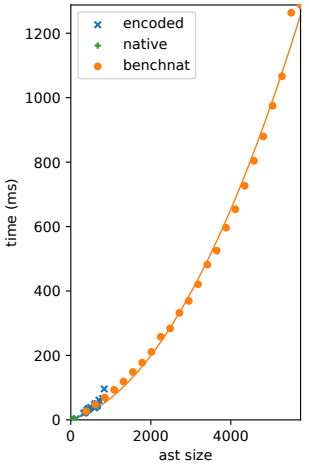
Figure 9a summarizes all examples used for evaluation. Sizes of examples are measured by the number of AST nodes. For those implemented using native data types, we further count QUAL(%) as the percentage of qualifier elements relative to the unqualified program sizes. For most examples,

	ENCODED		NATIVE		
TESTCASE	SIZE	TIME (MS)	SIZE	QUAL(%)	TIME (MS)
PAIR-TRANS	663	50.50	30	0	0.25
PAIR-OPAQUE	666	45.10	33	0	0.35
PAR-VAR1	437	33.00	45	10	0.90
PAR-VAR2	437	35.40	45	10	0.85
PAR-FUN1	322	20.65	49	11	1.30
PAR-FUN2	363	23.35	53	10	1.25
PAR-REF1	446	34.95	54	8	1.05
PAR-REF2	446	34.70	54	8	1.05
SEQ-CTR	586	37.95	66	3	1.50
PAR-CTR1	651	42.90	65	7	1.75
PAR-CTR2	606	38.00	57	8	1.25
PAR-SHR1	409	30.00	56	17	1.35
PAR-SHR2	409	30.20	56	17	1.10
WITHF-ESC	132	4.00	35	59	0.45
WITHF-SEQ	599	46.55	76	21	1.65
WITHF-PAR	704	61.30	103	20	2.85
LIST-SUM	836	95.95	80	3	2.15
LIST-MAP	797	66.50	94	22	2.40
MLIST-SEP	606	50.15	—	—	—
MLIST-SHR	495	38.65	—	—	—

(a) Example statistics.



(b) Scaling on examples.



(c) Scaling on ChurchNat.

Fig. 9. Example statistics and performance scaling results.

this is in the range 0–20%, showing a moderate annotation burden; the WITHF-ESC example is an outlier, mainly because the example has simple behavior but carries a nontrivial argument signature. We note that annotations are concentrated at function definitions; in larger programs where each function is called multiple times, the annotation percentage would be further amortized.

Time checking each example (TIME) is measured by the average of 20 consecutive runs.[6] We plot the checking time of individual examples in Figure 9b. To evaluate how $G^{\blacklozenge}_{\Leftrightarrow}$ scales on larger inputs, we use synthetic examples of Church-encoded natural numbers, shown in Figure 9c; the runtime fits a quadratic trend as both term sizes and context depths increase. The same quadratic curve, overlaid in Figure 9b, shows an imperfect but close match.

5.3 Implicit Type-and-Qualifier Instantiation

While $G^{\blacklozenge}_{\Leftrightarrow}$ does not infer type-and-qualifier instantiations, its qualifier inference for functions already adopts an eager instantiation approach inspired by Dunfield and Krishnaswami [22], resolving lower bound constraints on qualifier holes. Supporting full instantiation inference would require a more general, constraint-based approach such as local type inference [43], and we identify two specific challenges. First, $G^{\blacklozenge}_{\Leftrightarrow}$ enjoys *impredicative* polymorphism, which is beyond the capability of the *complete and easy* approach [22] and requires more sophisticated solutions [14, 57]. Second, the interleaving of types and qualifiers introduces upper bound constraints on qualifier variables, requires eliminating ill-scoped qualifier variables from type constraints, and demands greatest lower bound computation for qualifiers. We believe the procedures developed in this work—specifically avoidance (Section 4.3) and qualifier exposure (Figure 5)—provide reusable foundations towards generalizing these into a constraint-solving framework.

5.4 Alternative Avoidance Design

Related works that track resources using type qualifiers also observe the avoidance problem, and we compare them as follows. Simplistically, some works [8, 24] replace variables using *top* qualifiers:

```
{ let b = ref 42; fun () => b }   //: (Unit → Ref[Int]{*})
```

Nevertheless, without names, they stop tracking the sharing of escaping resources, but only that *there are some resources*. This suffices for tracking resources with scoped lifetimes.

Recent work [56] proposes refining top qualifiers using existential types. This choice works for their setup, but appears unideal for tracking reachability in more general settings. Illustrated below:

```
// extRes: ∃ s <:♦. Container[Resource^s]
let (a, resA) = extRes  // [···, a <:♦] ⊢ resA  ⇒  Container[Resource^a]^resA
let (b, resB) = extRes  // [···, b <:♦] ⊢ resB  ⇒  Container[Resource^b]^resB
```

If `extRes` is unpacked multiple times, the resulting `a` and `b` still represent the same reachability, but now have individual identities. This complicates later uses of `resA` and `resB`, where the type system needs to reason about their sharing. In contrast, using self-references, different *unpacked* instances of the same resource share the identity of the original resource:

```
// extRes: μs.Container[Resource^s]♦
let resA = extRes  // [···] ⊢ resA  ⇒  Container[Resource^extRes]^resA
let resB = extRes  // [···] ⊢ resB  ⇒  Container[Resource^extRes]^resB
```

6 Related Work

Tracking Reachability/Capturing in Types. This work is closely related to the original development of reachability types [6] and its polymorphic variant $F^{\blacklozenge}_{<:}$ [53], both of which have borrowed ideas from separation logic [39, 46]. Bao et al. [5] then established a logical relation model for

[6] Measured using Linux 6.6.87.2 on WSL 2.6.3.0 with Intel Ultra 7 258V.

a monomorphic substrate of $F^{\blacklozenge}_{<:}$, and Deng et al. [16] extended $F^{\blacklozenge}_{<:}$'s store model for added expressiveness. Our development builds on the logical relation model [5], adapted to match the terminating substrate of Deng et al. [16]. Nevertheless, to address the shared limitation from prior work that their subtyping does not allow conversions involving self-reference, our semantic model deviates from Bao et al. [5] in nontrivial ways, by interpreting locations reachable from values in a type-dependent manner instead of leaving them invariant. Moreover, this work presents a mechanized algorithmic development, which has no prior counterpart.

Closely related, capturing types [8] integrate escape checking and capability tracking into Scala 3, drawing on modal type theory [34] and using boxing to achieve capture tunneling with polymorphism. Both capturing types and reachability types track named resources via variable names, but differ in how they handle unnamed resources, as discussed in Section 5.4. Early capturing types use a top qualifier for all unnamed resources, which suffices for escape detection but carries no concrete information for separation reasoning. *Degrees of separation* [55] refines this with read/write distinctions at the cost of extra annotations, while freshness markers in reachability types [53]—and in this work—represent unique resources directly but flag any sharing regardless of effect. *Reach capabilities* [56] further refine top qualifiers with existentials to track internal resources, sharing motivation with our self-references, but, as discussed in Section 5.4, may complicate identity and separation reasoning. To our knowledge, no formal presentation has combined reach capabilities with separation checks; our self-references continue to support them.

Ownership Types. Originally developed for object-oriented programming, ownership type systems [11, 12, 35, 45] control the access paths to objects and enforce heap invariants. Many extensions have been developed on top of ownership types, such as disjointness domains [9] to express local alias invariants and external uniqueness [13] to relax the uniqueness restriction, among others. While ownership types track sophisticated properties, they usually require considerable annotation effort. As a remedy, ownership inference tools [18, 27] have been developed, combining points-to analysis, constraint solving, and human interaction to achieve practical results. As a prominent example, Rust [30, 33] implements a strong ownership model following the "shared XOR mutable" principle and has seen wide adoption in systems programming.

Similarly, reachability types aim at regulating aliases in higher-order languages. Without a primitive notion of *ownership*, they track sharing and separation via reachability qualifiers. While this suffices to achieve some access control, patterns like uniqueness can only be realized by layering an additional effect system [17]. By separating effects and requiring an acyclic heap structure, inference of reachability types is relatively straightforward, as demonstrated by $G^{\blacklozenge}_{\Leftrightarrow}$ in this work.

The Avoidance Problem. Systems with dependent types often encounter *the avoidance problem*: a term's type may mention a variable that has gone out of scope, and no minimal supertype avoiding that variable exists. The problem does not arise in systems with full dependent types, where substitution is unrestricted; it appears in systems where variables can only be substituted by other variables, such as DOT [47], bounded existentials [26], and ML-style modules. Consequently, some designs of ML-style module systems lack complete type checking [31]; refined designs avoid the problem by elaborating escaped variables into implicit existential types [20, 48]. In this work, variable escaping is an instance of the avoidance problem, and our solution, the self-reference, plays a similar role to implicit existential types in ML-style module systems. However, self-references are not existential types, as they do not require term-level packing/unpacking and can be smoothly introduced via subtyping. Their subtyping also differs from that of bounded existential types [52].

Self-References are not Recursive Types. While we borrow the μ-notation, self-references differ from recursive types [1] in several respects. Both let a construct refer to itself, but a self-reference

refers to a *term*—specifically, a dependent variable naming the value being typed—whereas recursive types refer to themselves at the type level. In the `Counter` example of Section 1, the self-reference stands for the counter value itself, and cannot be recursively expanded the way a recursive type unfolds into a copy of itself. Qualifiers are also finite sets: self-references *can* be unfolded by subtyping, but such unfolding saturates, so we never face the infinite equivalent or isomorphic qualifiers that make subtyping of recursive types subtle [59]. Moreover, when a self-reference is unfolded, a function value of the desired qualifier is guaranteed to be inhabited, avoiding the bad-bounds unsoundness familiar from DOT-like systems [4]. We leave extending reachability types with genuine recursive types as future work.

Type Inference with Subtyping. The Hindley-Milner (HM) typing algorithm [15] is able to infer principal types in polymorphic languages without subtyping. HM(X) [37] extends HM to constrained types including subtyping while preserving principal types. More recent approaches aim for compactness, inferring more compact types in the presence of subtyping [19, 41]; Parreaux and Chau [42] further integrates a set-algebraic structure into the type system, which can be extended to track non-escaping region variables [24].

As opposed to global inference algorithms, bidirectional typing [21, 23] features local reasoning at the cost of some explicit type annotations. Our adaptation for $\mathrm{G}^{\blacklozenge}_{<:}$ layers qualifiers on top of bidirectional typing: we infer qualifiers via an eager instantiation algorithm in the style of Dunfield and Krishnaswami [22], and employ a hybrid checking/synthesizing mode similar to the refinement strengthening of Polikarpova et al. [44]. Type and qualifier *instantiations* at polymorphic call sites, however, remain explicit. Inferring them is non-trivial: $\mathrm{G}^{\blacklozenge}_{\Leftrightarrow}$ supports impredicative polymorphism, for which implicit instantiation can render subtyping undecidable [10] and lies beyond the *complete-and-easy* approach [22], calling for more sophisticated constraint-based solutions [14, 38, 43, 57]; moreover, the interleaving of types and qualifiers introduces additional scoping and bound-computation concerns. We leave this as future work.

7 Conclusion

In this work, we investigated expressive subtyping with self-references and inference of qualifiers in reachability types. We identified the limitations of prior work and proposed a new declarative reachability type system $\mathrm{G}^{\blacklozenge}_{<:}$ with the combined notion of type-and-qualifier subtyping, enabling sound specification of the subtyping behavior of self-references. We also developed the first bidirectional typing algorithm $\mathrm{G}^{\blacklozenge}_{\Leftrightarrow}$ for reachability types and proved that it is decidable and sound with respect to $\mathrm{G}^{\blacklozenge}_{<:}$. To evaluate the expressiveness of our algorithm, we implemented and tested a variety of examples involving data structures. Both the declarative and algorithmic systems are mechanized in Lean. Together, they advance the usability of reachability types, and we believe they represent major steps towards practical programming languages.

Data Availability Statement

The extended version of this work can be found in Jia et al. [29]. Our artifact is available on Zenodo [28] and https://github.com/TiarkRompf/reachability/tree/main/checking/lean_v2.

Acknowledgments

We thank Haotian Deng for related development on reachability types. We thank Zhe Zhou, Patrick LaFontaine, Craig Liu, and Yueyang Tang for their feedback on early drafts. We thank anonymous reviewers for their valuable comments. This work was supported in part by NSF award 2348334 and an Augusta University faculty startup package, as well as gifts from Meta, Google, Microsoft, and VMware.

References

[1] Roberto M. Amadio and Luca Cardelli. 1993. Subtyping Recursive Types. *ACM Trans. Program. Lang. Syst.* 15, 4 (1993), 575–631. doi:10.1145/155183.155231

[2] 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 (Lecture Notes in Computer Science, Vol. 9600)*. Springer, 249–272. doi:10.1007/978-3-319-30936-1_14

[3] Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *POPL*. ACM, 666–679. doi:10.1145/3009837.3009866

[4] Nada Amin and Ross Tate. 2016. Java and scala's type systems are unsound: the existential crisis of null pointers. In *OOPSLA*. ACM, 838–848. doi:10.1145/2983990.2984004

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

[6] 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 (2021), 1–32. doi:10.1145/3485516

[7] Clement Blaudeau, Didier Rémy, and Gabriel Radanne. 2025. Avoiding Signature Avoidance in ML Modules with Zippers. *Proc. ACM Program. Lang.* 9, POPL (2025), 1962–1991. doi:10.1145/3704902

[8] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.* 45, 4 (2023), 21:1–21:52. doi:10.1145/3618003

[9] Stephan Brandauer, Dave Clarke, and Tobias Wrigstad. 2015. Disjointness domains for fine-grained aliasing. In *OOPSLA*. ACM, 898–916. doi:10.1145/2814270.2814280

[10] Jacek Chrzaszcz. 1998. Polymorphic Subtyping Without Distributivity. In *MFCS (Lecture Notes in Computer Science)*. Springer, 346–355. doi:10.1007/BFB0055784

[11] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming*. Lecture Notes in Computer Science, Vol. 7850. Springer, 15–58. doi:10.1007/978-3-642-36946-9_3

[12] David Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. ACM, 48–64. doi:10.1145/286936.286947

[13] Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *ECOOP (Lecture Notes in Computer Science, Vol. 2743)*. Springer, 176–200. doi:10.1007/978-3-540-45070-2_9

[14] Chen Cui, Shengyi Jiang, and Bruno C. d. S. Oliveira. 2023. Greedy Implicit Bounded Quantification. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 2083–2111. doi:10.1145/3622871

[15] Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *POPL*. ACM Press, 207–212. doi:10.1145/582153.582176

[16] 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 (2025), 3398–3425. doi:10.1145/3763172

[17] Haotian Deng, Siyuan He, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2025. Free to Move: Reachability Types with Flow-Sensitive Effects for Safe Deallocation and Ownership Transfer. *CoRR* abs/2510.08939 (2025). doi:10.48550/ARXIV.2510.08939

[18] Werner Dietl, Michael D. Ernst, and Peter Müller. 2011. Tunable Static Inference for Generic Universe Types. In *ECOOP (Lecture Notes in Computer Science, Vol. 6813)*. Springer, 333–357. doi:10.1007/978-3-642-22655-7_16

[19] Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. In *POPL*. ACM, 60–72. doi:10.1145/3009837.3009882

[20] Derek Dreyer, Karl Crary, and Robert Harper. 2003. A type system for higher-order modules. In *POPL*. ACM, 236–249. doi:10.1145/604131.604151

[21] Jana Dunfield and Neel Krishnaswami. 2022. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2022), 98:1–98:38. doi:10.1145/3450952

[22] Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*. ACM, 429–442. doi:10.1145/2500365.2500582

[23] Jana Dunfield and Frank Pfenning. 2004. Tridirectional typechecking. In *POPL*. ACM, 281–292. doi:10.1145/964001.964025

[24] Cunyuan Gao and Lionel Parreaux. 2025. A Lightweight Type-and-Effect System for Invalidation Safety: Tracking Permanent and Temporary Invalidation with Constraint-Based Subtype Inference. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 2623–2653. doi:10.1145/3763144

[25] Giorgio Ghelli. 1995. Divergence of F< Type Checking. *Theor. Comput. Sci.* 139, 1&2 (1995), 131–162. doi:10.1016/0304-3975(94)00037-J

[26] Giorgio Ghelli and Benjamin C. Pierce. 1998. Bounded Existentials and Minimal Typing. *Theor. Comput. Sci.* 193, 1-2 (1998), 75–96. doi:10.1016/S0304-3975(96)00300-3

[27] Wei Huang, Werner Dietl, Ana L. Milanova, and Michael D. Ernst. 2012. Inference and Checking of Object Ownership. In *ECOOP (Lecture Notes in Computer Science)*. Springer, 181–206. doi:10.1007/978-3-642-31057-7_9

[28] Songlin Jia, Guannan Wei, Siyuan He, Yuyan Bao, and Tiark Rompf. 2026. *Escape with Your Self: Sound and Expressive Bidirectional Typing with Avoidance for Reachability Types (Artifact)*. doi:10.5281/zenodo.19340767

[29] Songlin Jia, Guannan Wei, Siyuan He, Yuyan Bao, and Tiark Rompf. 2026. Escape with Your Self: Sound and Expressive Bidirectional Typing with Avoidance for Reachability Types (Extended Version). doi:10.48550/arXiv.2404.08217

[30] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *Commun. ACM* 64, 4 (2021), 144–152. doi:10.1145/3418295

[31] Xavier Leroy. 2000. A modular module system. *J. Funct. Program.* 10, 3 (2000), 269–303. doi:10.1017/S0956796800003683

[32] Mark Lillibridge. 1996. *Translucent Sums: A Foundation for Higher-Order Module Systems.* Ph. D. Dissertation.

[33] Nicholas D. Matsakis and Felix S. Klock II. 2014. The rust language. In *HILT*. ACM, 103–104. doi:10.1145/2663171.2663188

[34] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Log.* 9, 3 (2008), 23:1–23:49. doi:10.1145/1352582.1352591

[35] James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *ECOOP (Lecture Notes in Computer Science)*. Springer, 158–185. doi:10.1007/BFB0054091

[36] Martin Odersky, Aleksander Boruch-Gruszecki, Edward Lee, Jonathan Immanuel Brachthäuser, and Ondrej Lhoták. 2022. Scoped Capabilities for Polymorphic Effects. *CoRR* abs/2207.03402 (2022). doi:10.48550/ARXIV.2207.03402

[37] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theory Pract. Object Syst.* 5, 1 (1999), 35–55. doi:10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4

[38] Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored local type inference. In *POPL*. ACM, 41–53. doi:10.1145/360204.360207

[39] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (Lecture Notes in Computer Science, Vol. 2142)*. Springer, 1–19. doi:10.1007/3-540-44802-0_1

[40] Leo Osvald, Grégory M. Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *OOPSLA*. ACM, 234–251. doi:10.1145/2983990.2984009

[41] Lionel Parreaux. 2020. The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl). *Proc. ACM Program. Lang.* 4, ICFP (2020), 124:1–124:28. doi:10.1145/3409006

[42] Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: principal type inference in a Boolean algebra of structural types. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 449–478. doi:10.1145/3563304

[43] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44. doi:10.1145/345099.345100

[44] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *PLDI*. ACM, 522–538. doi:10.1145/2908080.2908093

[45] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. 2006. Generic ownership for generic Java. In *OOPSLA*. ACM, 311–324. doi:10.1145/1167473.1167500

[46] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74. doi:10.1109/LICS.2002.1029817

[47] Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *OOPSLA*. ACM, 624–641. doi:10.1145/2983990.2984008

[48] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2014. F-ing modules. *J. Funct. Program.* 24, 5 (2014), 529–607. doi:10.1017/S0956796814000264

[49] Lukas Rytz and Martin Odersky. 2012. Relative Effect Declarations for Lightweight Effect-Polymorphism. (2012). http://infoscience.epfl.ch/record/175546

[50] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6 (2024), 40:1–40:75. doi:10.1145/3676954

[51] Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: adding reference immutability to Java. In *OOPSLA*. ACM, 211–230. doi:10.1145/1094811.1094828

[52] Stefan Wehr and Peter Thiemann. 2011. On the Decidability of Subtyping with Bounded Existential Types and Implementation Constraints. *New Gener. Comput.* 29, 1 (2011), 87–124. doi:10.1007/S00354-010-0100-1

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

[54] Anxhelo Xhebraj, Oliver Bracevac, Guannan Wei, and Tiark Rompf. 2022. What If We Don't Pop the Stack? The Return of 2nd-Class Values. In *ECOOP (LIPIcs, Vol. 222)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:29. doi:10.4230/LIPICS.ECOOP.2022.15

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

[56] Yichen Xu, Oliver Bracevac, Cao Nguyen Pham, and Martin Odersky. 2025. What's in the Box: Ergonomic and Expressive Capture Tracking over Generic Data Structures. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 1726–1753. doi:10.1145/3763112

[57] Jinxu Zhao and Bruno C. d. S. Oliveira. 2022. Elementary Type Inference. In *ECOOP (LIPIcs, Vol. 222)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:28. doi:10.4230/LIPICS.ECOOP.2022.2

[58] Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. A mechanical formalization of higher-ranked polymorphic type inference. *Proc. ACM Program. Lang.* 3, ICFP (2019), 112:1–112:29. doi:10.1145/3341716

[59] Litao Zhou, Yaoda Zhou, and Bruno C. d. S. Oliveira. 2023. Recursive Subtyping for All. *Proc. ACM Program. Lang.* 7, POPL (2023), 1396–1425. doi:10.1145/3571241

[60] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. 2010. Ownership and immutability in generic Java. In *OOPSLA*. ACM, 598–617. doi:10.1145/1869459.1869509

Received 2025-11-14; accepted 2026-04-03