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

GUANNAN WEI, Purdue University, USA
OLIVER BRAČEVAC*, Purdue University, USA and Galois, Inc., USA
SONGLIN JIA, Purdue University, USA
YUYAN BAO, Augusta University, USA
TIARK ROMPF, Purdue University, USA

Fueled by the success of Rust, many programming languages are adding substructural features to their type systems. The promise of tracking properties such as lifetimes and sharing is tremendous, not just for low-level memory management, but also for controlling higher-level resources and capabilities. But so are the difficulties in adapting successful techniques from Rust to higher-level languages, where they need to interact with other advanced features, especially various flavors of functional and type-level abstraction. What would it take to bring full-fidelity reasoning about lifetimes and sharing to mainstream languages? Reachability types are a recent proposal that has shown promise in scaling to higher-order but monomorphic settings, tracking aliasing and separation on top of a substrate inspired by separation logic. However, naive extensions on top of the prior reachability type system λ^* with type polymorphism and/or precise reachability polymorphism are unsound, making λ^* unsuitable for adoption in real languages. Combining reachability and type polymorphism that is precise, sound, and parametric remains an open challenge.

This paper presents a rethinking of the design of reachability tracking and proposes new polymorphic reachability type systems. We introduce a new freshness qualifier to indicate variables whose reachability sets may grow during evaluation steps. The new system tracks variables reachable in a single step and computes transitive closures only when necessary, thus preserving chains of reachability over known variables that can be refined using substitution. These ideas yield the simply-typed $\lambda^{\blacklozenge}$-calculus with precise lightweight, *i.e.*, quantifier-free, reachability polymorphism, and the $\mathsf{F}^{\blacklozenge}_{<:}$-calculus with bounded parametric polymorphism over types and reachability qualifiers, paving the way for making true tracking of lifetimes and sharing practical for mainstream languages. We prove type soundness and the preservation of separation property in Coq. We discuss various applications (*e.g.*, safe capability programming), possible effect system extensions, and compare our system with Scala's capture types.

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

Additional Key Words and Phrases: type systems, reachability types, polymorphism, aliasing, effects

ACM Reference Format:
Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proc. ACM Program. Lang.* 8, POPL, Article 14 (January 2024), 32 pages. https://doi.org/10.1145/3632856

*Work completed while at Purdue University

Authors' addresses: Guannan Wei, Purdue University, West Lafayette, IN, USA, guannanwei@purdue.edu; Oliver Bračevac, Purdue University, West Lafayette, IN, USA and Galois, Inc., Portland, OR, USA, oliver@galois.com; Songlin Jia, Purdue University, West Lafayette, IN, USA, jia137@purdue.edu; Yuyan Bao, Augusta University, USA, yubao@augusta.edu; Tiark Rompf, Purdue University, USA, tiark@purdue.edu.

ACM 2475-1421/2024/1-ART14
https://doi.org/10.1145/3632856

1 INTRODUCTION

Type systems based on ownership and borrowing are seeing increasing practical adoption, most prominently for ensuring memory safety in comparatively low-level "systems languages" such as Rust [Matsakis and Klock 2014]. But what about higher-level languages, specifically those that rely on a larger degree on functional and type-level abstraction (*e.g.*, Scala and OCaml)?

Tracking substructural properties such as lifetimes and sharing in the type system holds great promise, not only for low-level memory management, but also for managing a variety of other resources (*e.g.*, files, network sockets, access tokens, mutex locks, etc.), for tracking effects (*e.g.*, via capabilities for exceptions, algebraic effects, continuations, callbacks via async/await, etc.), as well as for compiler optimizations (*e.g.*, fine-grained dependency analysis [Bračevac et al. 2023], safe destructive updates, etc.). Therefore, it is no surprise that several mainstream languages are moving in this direction with experimental proposals backed by serious engineering efforts, which are to a large degree inspired by the success of Rust, *e.g.*, Linear Haskell [Bernardy et al. 2018] and Scala capture types [Boruch-Gruszecki et al. 2023].

However, these proposals all focus on relatively narrow substructural properties rather than attempting to model lifetimes and sharing with similar generality as Rust's ownership and borrowing approach. For example, Linear Haskell specifically tracks multiplicity of uses, and Scala capture types specifically target effect capabilities. Of course, this is neither neglect nor coincidence, but the observable effect of an underlying hard problem: ownership type systems [Clarke et al. 2013, 1998; Noble et al. 1998] that would enable tracking more sophisticated lifetime properties traditionally rely on strict heap invariants (selectively relaxed via borrowing [Hogg 1991]) that are difficult to enforce in the presence of pervasive functional and type-level abstraction (*e.g.*, Figure 1).

In this paper, we build on *reachability types* [Bao et al. 2021], a recent proposal for bringing the benefits of ownership type systems to higher-order languages, in a way that is more directly inspired by separation logic [O'Hearn et al. 2001; Reynolds 2002]. We propose new reachability type systems that are suitable for tracking aliasing and separation in *polymorphic* higher-order languages, addressing several limitations of Bao et al.'s work, and thus paving the way for adoption in realistic languages.

Reachability Types: Tracking Sharing. The key idea of reachability types is to track reachable variables/locations as type qualifiers, which is best demonstrated by an example with ML-style references (types and contexts shown as comments):

```
val x = new Ref(0)  // : Ref[Int]^x in context [ x: Ref[Int]♦ ]
val y = x           // : Ref[Int]^y in context [ y: Ref[Int]^x, x: Ref[Int]♦ ]
```

Variable `x` is bound to a *freshly* allocated reference, indicated by the freshness marker ♦ which is part of the new design in this paper. The assigned type qualifier of expression `x` tracks only `x` itself, and just the same, the alias `y` of `x` tracks only `y` itself. By inspecting the typing context, we are able to collect all reachable variables transitively on demand, *e.g.*, through `y` we can reach both itself and `x`.

We say that variable `x` is *reachable* from `y` in the sense that there is an access path starting from `y` to `x`, corresponding to their reachability in the runtime store. Note that reachability is not symmetric, and stronger than aliasing, *e.g.*, `y` reaches `x`, but `x` does not reach `y`. Reachability is cheaper to compute than full aliasing, yet sufficient to check non-aliasing, *i.e.*, separation.

Qualifiers for function types include the free variables captured by the function from its defining scope, reflecting the fact that these free variables are *reachable* by the runtime closure value of a function. For example, function `inc` below captures variable `counter` defined in the outer scope:

```
val counter = new Ref(0)
def inc(n: Int): Unit = { counter := !counter + n } // : inc: (Int => Unit)^counter
```

```
def counter(n: Int) = {                  // counter: Int => μp.Pair[(()=>Unit)^{p}, (()=>Unit)^{p}]♦
  val c = new Ref(n)                     //   : Ref[Int]^{c}
  (() => c += 1, () => c -= 1)           //   : Pair[(()=>Unit)^{c}, (()=>Unit)^{c}]^{c}
}
                                         // instantiate the self-reference p with bound name ctr:
val ctr = counter(0)                     //   : Pair[(()=>Unit)^{ctr}, (()=>Unit)^{ctr}]^{ctr}
                                         // name ctr abstracts over its captured variables:
val incr = fst(ctr)                      //   : (()=>Unit)^{ctr}
val decr = snd(ctr)                      //   : (()=>Unit)^{ctr}
```

Fig. 1. An example (adapted from [Bao et al. 2021]) demonstrating first-class functions supported by reachability types. The `counter` function returns two closures over a shared mutable reference (which is a fresh value before binding it to `c`). The return value is a pair typed with a self-reference `p` to express the capture of `c` by both closures. The self-reference introduced by the μ-notation is similar to DOT, though directly attached to function types in our formalization (cf. Section 8.1 for the encoding and Section 4.1 for the formal syntax). In comparison, Rust's type system prevents returning closures over local mutable references due to its "shared XOR mutable" restriction, and requires falling back to dynamic reference counting for similar functionality.

The qualifier of a function can be considered as the *observability* of the function (or short, its *ability*), since it constrains what part of the context can be used within the function body. In addition, both function arguments and return types can be qualified with reachability sets.

Reachability Types: Tracking Separation. The idea of tracking reachability at the type level gives rise to powerful reasoning capabilities — most importantly, when considering the absence of reachability, namely *separation*. Two terms are separate when their type qualifiers are (transitively) disjoint. Argument qualifiers of function types indicate the permissible overlap between a call-site argument and the function's reachable set. Consider the following `inc` function capturing `counter`:

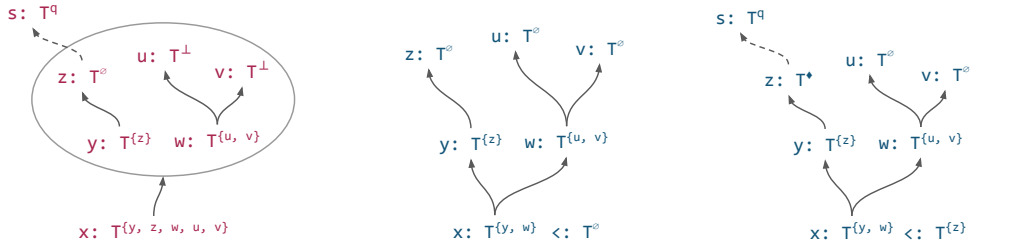
```
def inc(r: Ref[Int]♦): Unit = { r := !counter + 1 }
```

Its argument qualifier is the fresh marker, demanding that the argument can be anything but cannot be aliased with the resources that the function can *observe* from its context (*i.e.*, cannot be `counter`).

The metatheory of reachability types guarantees not only preservation of types but also preservation of separation: if two expressions have disjoint qualifiers, they will evaluate to disconnected object graphs at runtime (Section 4.4.3). Taking reachability and separation as the fundamental building blocks of a type system stands in contrast to traditional ownership type systems that put heap invariants about unique access paths first and selectively re-introduce sharing via borrowing. Crucially, reachability and separation appear as more fundamental properties in the sense that formal accounts of Rust's type system [Jung et al. 2018] are typically expressed using separation logic as the metalanguage.

Escaping Closures with Shared Mutable Data. Previous work [Bao et al. 2021] has shown how reachability types elegantly support functional abstraction beyond what is available in Rust. For example, Figure 1 shows a program with escaping functions that can track the sharing of locally-defined resources, which cannot be expressed under Rust's "shared XOR mutable" constraint.

In Figure 1, we define the `counter` function that returns a pair of functions to increment or decrement a mutable variable. Both functions *capture* the local heap-allocated reference cell `c`. Hence, before returning, both components of the pair track qualifier `c`. Once *escaped* from `c`'s defining scope, the name `c` is not meaningful in the outer scope, but we still need to track the sharing between the pair components. Reachability types preserve the tracking of shared resources through *self-references*, a concept borrowed from Dependent Object Types (DOT) [Amin et al. 2016; Rompf and Amin 2016].



(a) λ^* [Bao et al. 2021] tracks all reachable variables transitively. Leaf nodes are untracked ($\bot$ in λ^*).

(b) $\lambda^\blacklozenge$ tracks one-step reachability by default. `x: T{y, w}` can be upcast to untracked `x: T∅`.

(c) $\lambda^\blacklozenge$ models freshness using the ♦ marker, which prevents further upcasting via subtyping (beyond `z`).

Fig. 2. Illustration and comparison of different reachability tracking mechanisms. We use solid lines for direct reachability, and dashed lines for reachability that is unobservable in the current context (cf. Section 3.1.2). 2a illustrates prior work by Bao et al. [2021], 2b reflects both this work and Scala capture types [Boruch-Gruszecki et al. 2023], and 2c illustrates the unique feature of this work, which prevents upcasting through "fresh" variables, and thus allows substituting fresh variables with *larger* (but observably separate) reachability sets during evaluation. Thus, this work subsumes the essential aspects of both λ^* (separation) and capture types (qualifier refinement using subtyping).

Self-references are identifiers introduced at the type level, but can only be used in qualifiers. A self-reference serves as an abstraction of a set of captured variables. For example, the `counter` function in Figure 1 returns a pair with the self-reference `p`, which by subtyping is an upper bound of the captured reference `c`, *i.e.*, `{c} <: {p}`. When leaving the scope (but before binding the pair to a variable), we use the self-reference to *pack* the encapsulated resources that are not visible outside:

$$\texttt{Pair[(()=>Unit)}^{\texttt{c}}\texttt{, (()=>Unit)}^{\texttt{c}}\texttt{]} \rightsquigarrow \mu\texttt{p.Pair[(()=>Unit)}^{\texttt{p}}\texttt{, (()=>Unit)}^{\texttt{p}}\texttt{]}$$

Then, when the returned value is bound to an identifier, we *unpack* the self-reference and instantiate it to the bound variable. Unpacking eliminates the self-reference and replaces it with a concrete local binding (*i.e.*, `ctr` in Figure 1):

$$\mu\texttt{p.Pair[(()=>Unit)}^{\texttt{p}}\texttt{, (()=>Unit)}^{\texttt{p}}\texttt{]} \rightsquigarrow \texttt{Pair[(()=>Unit)}^{\texttt{ctr}}\texttt{, (()=>Unit)}^{\texttt{ctr}}\texttt{]}$$

Therefore, we have properly maintained the mutable sharing between the two pair components. In contrast, Rust does not allow two functions to capture shared resources in mutable ways, unless using dynamic reference counting or unsafe mechanisms to bypass the static ownership discipline.

In Section 8.1, we study the Church-encoding of pairs, and explain how function self-references support escaping values. It is important to note that, just like with recursive types [Amadio and Cardelli 1993; Zhou et al. 2023, 2022], the theory of subtyping for self references is subtle, and also differs from recursive types in important ways (*e.g.*, in having to preserve the identity of references). Hence, while our formalization (Sections 4 and 5) tracks self-references precisely in type assignment, it curbs their use in subtyping, and leaves a full development of subtyping for self references to future work. Our encodings of data types thus rely on coercions via η-expansion. However, these conversions justify corresponding subtyping rules that could be soundly added to implementations with first-class datatype support.

Preserving Chains of Reachability. As mentioned in the first motivating example, it suffices to assign a reflexive qualifier when typing variables, *i.e.*, `x` is typed as $\texttt{T}^{\texttt{x}}$ under a well-formed context. When necessary, we can compute the transitively saturated reachability sets on demand, *e.g.*, before computing intersections to check separation (see Section 3.1.5).

As an important new ingredient in this paper, this *one-step reachability* preserves the chains of reachability and maintains higher precision across substitution, both as part of dependent

function application and during reduction steps. This approach yields a new notion of "maybe-tracked" values, whose tracking status contextually and transitively depends on other variables from the context, *e.g.*, by refining reachability through the subtyping relation (see Section 3.1.4), x's reachability set in Figure 2b can be "upcast" to the empty set, reflecting its true untracked status.

Compared with the prior work, qualifiers in Bao et al. [2021]'s λ^*-calculus are assigned to always include all transitively reachable variables, *i.e.*, the reachability sets are eagerly saturated. The two different mechanisms are illustrated in Figure 2. In Figure 2a, λ^* tracks all variables that can be transitively reached from `x`, whereas in Figure 2b, `x` only tracks its immediate reachable variables, namely `{y, w}`. Tracking saturated reachability is not always necessary and leads to precision loss when the reachability set of a variable is refined to a smaller set using substitution. If the reachability set containing the variable is transitively saturated, the now superfluous elements cannot be removed, unless one would recompute the transitive closure from scratch.

Distinguishing Untracked, Tracked, and Fresh Resources. Another key limitation of Bao et al.'s λ^* is the use of an explicit *untracked* marker $\perp$ (*e.g.*, for pure functions and non-resource values) and reserving variable sets for both "fresh" (using $\varnothing$) and tracked resources. This design leads to a loss of tracking precision and interacts badly with type polymorphism (cf. [Wei et al. 2023], Appendix A). We found that it is more natural and intuitive to let qualifiers uniformly be sets, and optionally include the novel freshness marker $\blacklozenge$ as an explicit representation of *statically* unobservable variables/locations which may materialize during evaluation (*e.g.*, from allocations). While the $\blacklozenge$ placeholder as such requires careful treatment in subtyping chains (*e.g.*, Figure 2c), it also enables a harmonious integration of reachability with type polymorphism.

Polymorphic Reachability Types. Based on one-step reachability and the novel freshness representation, this paper proposes new variants of reachability types that track fine-grained lifetime properties for higher-order, imperative, and polymorphic languages.

We develop the $\lambda^{\blacklozenge}$-calculus (Section 4), featuring precise lightweight reachability polymorphism (without explicit quantifiers), deep dependencies in qualifier-dependent applications, and nested references. The $\lambda^{\blacklozenge}$-calculus overcomes fundamental expressiveness limitations of Bao et al.'s λ^*. Furthermore, on top of the $\lambda^{\blacklozenge}$-calculus, we develop extensions with bounded quantification over types and qualifiers (Section 5), leading to the $\mathsf{F}^{\blacklozenge}_{<:}$-calculus that for the first time can soundly support polymorphic data types with granular reachability tracking of components.

Contributions. We make the following specific contributions:

- We informally introduce the polymorphic reachability type system with several motivating examples (Section 2), highlighting key use cases and demonstrating expressiveness.
- We introduce the key new ideas that lead to precise reachability tracking (Section 3). Specifically, we propose a new mechanism that preserves transitive chains of reachability based on an explicit "freshness" representation.
- We present the formal theory and metatheory of (1) the $\lambda^{\blacklozenge}$-calculus with precise reachability polymorphism that improves over Bao et al. [2021]'s λ^*-calculus (Section 4), and (2) the $\mathsf{F}^{\blacklozenge}_{<:}$-calculus with bounded type-and-qualifier abstraction as an $\mathsf{F}_{<:}$-style extension of $\lambda^{\blacklozenge}$ (Section 5). We prove type soundness and the preservation of separation property for both calculi, which have been mechanized in Coq, along with the examples in this paper.
- We discuss decidability issues and our prototype type checker for $\mathsf{F}^{\blacklozenge}_{<:}$ (Section 6).
- We discuss limitations and possible extensions of the current system, including general nested references and flow-sensitive effect systems (Section 7).
- We present two case studies, the Church-encoding of polymorphic data types and a comparison with Scala capture types (Section 8). Our system subsumes both the original reachability types λ^* [Bao et al. 2021] and the essence of capture types [Boruch-Gruszecki et al. 2023].

Section 9 discusses related work and Section 10 concludes the paper. The Coq mechanization and prototype type checker can be found at https://github.com/tiarkrompf/reachability.

2 MOTIVATING EXAMPLES

In this section, we present several examples to demonstrate the expressiveness and applications of polymorphic reachability types. In these examples, we adopt a Scala-like surface syntax of $\lambda^{\blacklozenge}/\mathsf{F}^{\blacklozenge}_{<:}$, which are formally introduced in Sections 4 and 5.

2.1 Examples Enabled by Tracking Reachability and Separation

Scoped Borrowing. We start by implementing Rust-style borrowing using reachability types. As a general pattern, borrowing grants unique access to a resource and temporarily disables other accesses to that resource. Following Bao et al. [2021], we can define a combinator `borrow`:

```
def borrow[A⋆, B⋆](x: A)(block: (A => B)⋆): B = block(x)
```

Type checking in reachability types ensures that `x` and `block` are (transitively) separate (see details in Section 3.1.5), therefore `block` can only access the borrowed resource via its argument. A client program can look like the following snippet:

```
val x = new Ref(42)
borrow(x) { y => ... /* can only access the reference cell via y, but not x */ }
```

Non-Escaping. Reachability types are useful to track lifetimes of resources/capabilities. Consider the `withFile` combinator that ensures an opened file handle will eventually be closed after it is used:

```
def withFile[T⋆](path: String)(block: (File⋆ => T^∅)⋆): T^∅ = {
  val file = openFile(path); val res = block(file)
  closeFile(file); res
}
```

For safety, we must also ensure that the file handle accessed by `block` does not escape. In reachability types, this is guaranteed by the empty qualifier ∅ of `block`'s return type in the definition of `withFile`. A client program that attempts to return a closure capturing the `file` handle is rejected:

```
withFile("a.txt") { file => ...
  { () => file.readLine() } // type error: since (Unit => String)^file ⋠: (Unit => String)^∅
}
```

This pattern of encoding non-escaping capabilities has broad applications, especially for preventing leaks of resources that are only valid within a bounded lifetime, such as file handles, sockets, locks, stack-allocated effect handlers, etc. In Section 8.2, we compare reachability types with capture types [Boruch-Gruszecki et al. 2023], a recent proposal for tracking effects as capabilities in Scala.

Automatic Memory Management. Tracking resources that are non-escaping is also useful for scope-based automatic memory management (ARM). If a resource does not escape, we can safely reclaim it before leaving the scope.

```
def f(n: Int): Unit = {
  val temp = new Ref(n) // heap-allocated reference cell
  ...
  free(temp)            // compiler can insert a call to free
}
```

In the above example, the compiler can insert a call to `free` that deallocates the reference cell on the heap, akin to Rust-style `drop` function that is automatically inserted at the end of the lifetime.

Safe Parallelization. We now discuss safe parallelization enabled by ensuring non-interference of two functions. Recall that function types are qualified with a set of variables, representing the resources/capabilities that the function can access. If the qualifiers of two functions are separate, they cannot share any resources, thus it is safe to run them in parallel.

Consider the following library function `par` (adapted from Bao et al. [2021]) that takes two thunks as arguments. By ensuring that the qualifiers of the two thunks are disjoint, the implementation of `par` can safely schedule their execution in parallel without interference. A client program using `par` must provide two functions that use disjoint sets of resources, *e.g.*:

```
def par(a: (() => Unit)♦)(b: (() => Unit)♦): Unit
val c1 = new Ref(0); val c2 = new Ref(0)
par { c1 += ... /* ok: operate on c1 only, cannot access c2 */ }
    { c2 -= ... /* ok: operate on c2 only, cannot access c1 */ }
```

2.2 Examples Enabled by Layering an Effect System

Although reachability types offer ways to track usage and lifetimes of capabilities, they are sometimes too coarse-grained. For example, two functions attempting to read the same mutable reference cell can still be executed in parallel if there are no interfering writes. With an effect system (Section 7.2), we can not only discern non-interference with a finer granularity, but also enable move semantics, safe deallocation, and more.

Flow-Sensitivity and Move Semantics. Bao et al. [2021] have discussed layering a flow-sensitive effect system on top of tracking reachability. The basic idea is to track the effect induced by aliased variables and maintain the aliases and their effects with respect to control-flow structure. This enables not only tracking effects and identifying non-interference more precisely, but also expressing move semantics (ownership transfer) and safe deallocation as *destructive* effects.

To first see a simple example, consider that we have aliased variables `x` and `y`. Reading the content of `y` induces a `read` effect on `y`, and transitively on `x` too, as noted in the comment:

```
val x = new Ref(42); val y = x
!y    // : Int @read(y) in context [ y: Ref[Int]^x, x: Ref[Int]♦ ]
```

A powerful effect supported by Bao et al. is the destructive "kill" effect (terminology borrowed from data-flow analysis). The kill effect takes execution order into account (*i.e.*, flow-sensitive) and applying it to a variable disables any future use via all aliases of this resource. This effectively expresses a wide range of behaviors, such as ownership transfer and uniqueness. For example, the language could offer a primitive `move` for Rust-style ownership transfer, which induces the kill effect on its argument and yields a fresh value that uniquely holds the underlying resource:

```
val x = new Ref(42)
val y = move(x)          // : Ref[Int]^y @kill(x) in context [ y: Ref[Int]♦, x: Ref[Int]♦ ]
```

After the ownership transfer, `y` uniquely holds the underlying resource, and reading or writing `x` is a compile-time error.

Safe Deallocation with Effects. Similar to `move`, we can extend our language with a `free` primitive for memory deallocation. The effect of `free` is to "kill" its argument, and any use of that resource induces a compile-time use-after-free error.

```
// free: (x: Ref[Int]♦) => Unit @kill(x)
val x = new Ref(42); val y = x
free(y)     // : Unit @kill(y) in context [ y: Ref[Int]^x, x: Ref[Int]♦ ]
!x + !y     // type error: both !x and !y reach a killed resource
```

Note that the `free` function has a latent effect which kills its argument. This latent effect is propagated to downstream computation that effectively prohibits using any alias of that argument to `free`.

The key idea behind our effect system extension is to deploy an effect quantale [Gordon 2021] that gives rise to flow-sensitivity via partial sequential effect composition. In Section 7.2, we discuss the effect system extension on top of the new polymorphic reachability types in this paper.

Optimizing Effectful Functional Languages. Compiler optimizations often rely on precisely tracking dependencies between statements, so that reordering, rewriting, or elimination of statements preserve the original semantics. Bračevac et al. [2023] have showed that an effect system extension on top of reachability types (Section 7.2) can enable such fine-grained dependency analysis for higher-order effectful programs. Here, we demonstrate an example that optimizes "write-after-write":

```
val x1 = new Ref(42)
val x2 = x1
val x3 = (x1 := 10)  // @write(x1), can be eliminated by compiler
val x4 = (x2 := 20)  // @write(x2)
```

In Bračevac et al. [2023], programs are represented as graphs where nodes (*e.g.* `x1`, `x2`, etc.) are statements and edges are dependencies. Through an effect-and-dependency analysis, the compiler is able to identify that (1) node `x3` and `x4` write to the same reference cell since `x1` and `x2` transitively reach the same resource, and (2) `x4` does not directly depend on the side effect performed by `x3` (known as anti-dependency or soft-dependency). That is, there is no direct dependency from `x4` to `x3`, thus it is sound to eliminate the statement `x3` when generating code. For more sophisticated optimizations involving function abstractions, readers are referred to Bračevac et al. [2023].

Towards Polymorphic Reachability Types. We see a wide range of applications of reachability types, especially in enabling safe and novel programming paradigms for effects and capabilities in higher-order languages. While variants of examples discussed in this section can already be expressed in Bao et al. [2021]'s monomorphic system, their precise typing requires polymorphism, which Bao et al. [2021]'s reachability type system cannot express. We discuss the new ideas to make reachability types smoothly integrate with type polymorphism next.

3 POLYMORPHIC REACHABILITY TYPES

We first propose the simply-typed $\lambda^{\blacklozenge}$-calculus, which features a new treatment of freshness and a fine-grained reachability assignment, leading to a well-behaved and precise notion of reachability polymorphism. Then we show that the ideas smoothly scale to the $\mathsf{F}^{\blacklozenge}_{<:}$-calculus with type-and-qualifier abstraction.

In Appendix A of the extended version of this paper [Wei et al. 2023], we review the prior reachability type system λ^* [Bao et al. 2021] and its limitations with regards to reachability polymorphism.

3.1 Precise Reachability Polymorphism in $\lambda^{\blacklozenge}$

3.1.1 One-Step Reachability Tracking. $\lambda^{\blacklozenge}$ keeps reachability sets minimal in type assignment and only computes transitive closures *on demand* (cf. Section 3.1.5), which ensures that we can preserve chains of reachability and refine elements in the chain later by substitution or subtyping. In contrast, Bao et al. [2021] use an *"eager"* strategy to track aliases: typing relations assign *saturated* qualifiers, *i.e.*, these qualifiers are large enough to include all transitively reachable variables.

The new system's "on-demand" and Bao et al. [2021]'s "eager" tracking strategies each treat variable bindings differently (typing context shown to the right of ⊣):

```
val x = alloc()  // : T{x} ⊣ x: T{♦}          val x = alloc()  // : T{x}
val y = x        // : T{y} ⊣ y: T{x}, x: T{♦}  val y = x        // : T{x, y}
```

The on-demand version (left) only assigns the one-step reachability set `{y}`. It can be scaled to the saturated set by subtyping (`{y} <: {x,y}`), which includes the subset relation. In the eager version (right), `y` reaches `{x, y}`, transitively including `x`'s reachability set from the context.

On-demand tracking preserves the chains of reachability in typing: during reduction steps, qualifiers in the chain can be replaced with smaller reachable sets, leading to an increase in precision via substitution.

3.1.2 Freshness Marker ♦. We model potential freshness by adding a marker ♦ to qualifiers, connecting static observability with evaluation. A type $T^{\{\blacklozenge\}}$ describes expressions which cannot reach the currently observable variables (thus *observable separation*), but they may reach unobservable variables, including new references. The prime example is the reduction of allocations:

```
alloc() // : Ref[Int]{♦}   reduces to   ℓ // : Ref[Int]{ℓ}, where ℓ is a fresh location value
```

Before reduction, `alloc()` is fresh, *i.e.*, it must be tracked but is not bound to a variable. Afterwards, we have a new and definitely known store location, which is considered not fresh, thus ♦ vanishes. The presence of ♦ indicates that reduction steps may grow the qualifier, and its absence indicates that they will not. Bao et al.'s track/untrack system assumes that any tracked qualifier might grow.

The ♦ marker also serves as a "contextual freshness" indicator for function parameters, *e.g.*, here is the reachability-polymorphic identity function in $\lambda^{\blacklozenge}$:

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

The type specifies that `id` (1) cannot observe anything about its context (∅), and (2) it accepts arguments that may reach any unobservable variables. Thus, the `id` function accepts `T` arguments with any qualifier and the function body can only observe a fresh argument.

Adjusting parameter qualifiers permits controlling the overlap between functions and their arguments, *e.g.*, consider variants of `id` which close over some variable `z` in the context:

```
def id2(x: T{♦}  ): T{x} = { val u = z; x }   // : ((x: T{♦}  ) => T{x}){z}
def id3(x: T{♦,z}): T{x} = { val u = z; x }   // : ((x: T{♦,z}) => T{x}){z}
def id4(x: T{z}  ): T{x} = { val u = z; x }   // : ((x: T{z}  ) => T{x}){z}
```

The qualifiers on the function type and the parameter specify the reachability information that the implementation can *observe* about its context (only `z` here), and about any given argument, respectively. It also says that the implementation is *oblivious* to anything it *cannot observe*. Function `id2` accepts arguments reaching anything that does not (directly or transitively) reach `z`. But `id3` permits `z` in the parameter's qualifier, effectively allowing any argument. Finally, `id4`'s parameter lacks the freshness marker, constraining arguments to be contextually non-fresh. That is, only observable arguments which reach at most `z` are allowed.

With the freshness marker, it is no longer necessary to use ⊥ to indicate untracked values. In $\lambda^{\blacklozenge}$, qualifiers of untracked values (*e.g.*, primitive values) are simply denoted by the empty set ∅.

3.1.3 Precise Reachability Polymorphism. Unlike its λ^* version (cf. [Wei et al. 2023], Appendix A), `id` is truly reachability polymorphic, as it properly preserves the tracking status of arguments:

```
id(42)          // : Int{x}[x↦∅] = Int∅  ← unbound and untracked
id(alloc())     // : T{x}[x↦♦] = T{♦}    ← unbound and tracked (fresh)
```

The key design difference here is having the ♦ marker in qualifiers to explicitly communicate (non-)freshness which is preserved by dependent application and substitution. Consider a function that mutates a captured reference cell and returns the argument. We annotate that the argument `x` is potentially aliased with the captured argument `c1` but apply the function with argument `c2`. $\lambda^{\blacklozenge}$

would not propagate such imprecision by tracking one-step reachability. The return type only tracks the argument `x`. When applying different arguments to the function, precise reachability is retained:

```
... // c1: T^{c1}, c2: T^{c2}
def foo(x: T^{c1,♦}): T^{x} = { c1 := !c1 + 1; x } // : ((x: T^{c1,♦}) => T^{x}){c1}
foo(c1)                                            // : T^{c1}
foo(c2)                                            // : T^{c2} ← precision retained
```

In contrast, in Bao et al. [2021]'s system, this potential alias is propagated to the return type qualifier and we cannot get rid of it even when applying with a non-overlapped argument `c2`: `foo(c2) : T^{c1, c2}`. The freshness marker also prevents typing the problematic `fakeid` function, since `{♦}` is not compatible with the result qualifier `{x}`:

```
def fakeid(x: T♦): T^{x} = alloc()              // type error: {♦} ⊀: {x}
```

3.1.4 Maybe-Tracked and Subtyping. With one-step reachability tracking, a novel notion of "maybe-tracked" status emerges in $\lambda^{\blacklozenge}$. For example, the tracking status of `Int^{x}` only depends on the reachability of `x`, and therefore `x` is "maybe" tracked:

```
val x = 42          // : Int^{x}          ← bound but untracked
id(x)               // : Int^{x} <: Int^∅ ← unbound and upcast via one-step reachability
```

Moreover, `Int^{x}` is equivalent to `Int^∅`, upcast by one-step reachability using $\lambda^{\blacklozenge}$'s subtyping relation. Chasing the typing assumptions, both `{x} <: ∅` and `∅ <: {x}` hold in the above context, which justifies the equivalence. This reasoning step uses a subtyping rule for looking up qualifiers of bound variables in the context (see Section 4.2.6), which permits smaller, context-dependent steps to form reachability chains *as long as qualifiers in the chain are all non-fresh.* Therefore, `id(y)` cannot be upcast since its one-step reachable variable `y` is fresh:

```
val y = alloc()     // : T^{y}           ← bound and tracked
id(y)               // : T^{y}           ← bound and cannot further upcast since y fresh
```

3.1.5 On-Demand Transitivity. When does the type system actually need to compute saturated qualifiers with the "on-demand" tracking strategy (Section 3.1.1)? Applying functions that expect fresh arguments is the only situation where this is necessary. For example, consider a function `f` that does not permit overlap between the argument's qualifier and its own reachable set:

```
val c1 = alloc()                  // : Ref[Int]^{c1} ⫞ c1: Ref[Int]♦
def f(x: Ref[Int]♦) = !c1 + !x    // : (f(x: Ref[Int]♦) => Int)^{c1}
val c2 = c1                       // : Ref[Int]^{c2}
f(c2)                             // type error: since {c1,c2} ∩ {c1} ≠ ∅
```

The application `f(c2)` should be rejected due to the lack of separation between `c2` and `f`. Since the one-step reachability strategy lets variable bindings reach only themselves by default, naively intersecting the function and argument at the call site would not detect that `c2` overlaps with `f` through `c1`. Thus, a sound overlap check at call sites must first compute *saturated* upper bounds on demand, and then compute their intersection. We discuss the formal details of saturated qualifiers and overlap checking further in Section 4.2.1.

Finally, it is worth noting that `c2`'s qualifier cannot be upcast through its reachability chain `c1` to `{♦}` via subtyping, which would result in unsound overlap checking (cf. Section 4.2.6).

3.1.6 Qualifier-Dependent Application. The $\lambda^{\blacklozenge}$-calculus also supports precise reachability polymorphism via dependent function applications. That is, given a function type $f(x : T_1^{q_1}) \to T_2^{q_2}$, the argument variable x may occur in the codomain type T_2 and qualifier q_2 (cf. Section 4.2.4). This is

more expressive than the λ^* system [Bao et al. 2021], which forbids occurrences within T_2 to ensure a sound treatment of escaping closures. Consider the following function returning another function:

```
val c = alloc()
def f(x: Ref[Int]^{c}) = () => x // : (f(x: Ref[Int]^{c}) => (Unit => Ref[Int]^{x})^{x})^∅
f(c)                             // : (Unit => Ref[Int]^{c})^{c}
```

We can assign the reachability set of `f`'s innermost return type, depending on the outer argument `x`. The dependent application `f(c)` yields a precise type, whereas the λ^*-calculus would have to upcast the returned function type to a self-reference before application (thus introducing imprecision). Such precision is enabled by $\lambda^\blacklozenge$'s refined freshness-marker model, which distinguishes fresh/growing from non-fresh/static qualifiers.

3.2 Type-and-Qualifier Abstractions in $\mathsf{F}^\blacklozenge_{<:}$

Next, we extend $\lambda^\blacklozenge$ with type-and-qualifier abstractions in the style of $\mathsf{F}_{<:}$ [Cardelli et al. 1994], resulting in the $\mathsf{F}^\blacklozenge_{<:}$-calculus. Such an extension is not possible on top of the λ^*-calculus, because of its confounding of fresh, tracked values and untracked values.

Type Abstractions. The first step towards $\mathsf{F}^\blacklozenge_{<:}$ is to add $\mathsf{F}_{<:}$-style quantification over proper types *without* qualifiers. This is already attractive and enough to express the identity function with *both* type and lightweight reachability polymorphism. The following definition of `id` adds the type parameter `T` and does not require $\mathsf{F}_{<:}$-style abstraction of qualifiers:

```
def id[T <: Top](x: T^♦): T^{x} = x
```

As in $\mathsf{F}_{<:}$, we add an upper bound `Top` of all types to the system. However, reachability sets attached to proper types must be concrete and cannot be abstracted over.

Qualifier Abstractions. We now introduce an *abstract qualifier* and an *upper bound qualifier* in the style of $\mathsf{F}_{<:}$. In this way, the polymorphic identity function is a shorthand notation that does not need to use the abstract qualifier. The fully desugared term is

```
def id[T^z <: Top^♦](x: T^♦): T^{x} = x
def id[T](x: T^♦) = x // shorthand notation
```

where `z` is the abstract qualifier variable bounded by ♦. One could further omit the abstract qualifier, type-and-qualifier bound, and return type using the shorthand notation shown above.

Although the additionally introduced abstract qualifier (`z`) does not yield further expressiveness for the identity function, quantified qualifiers vary independently of the type variable, and one is free to attach them to any proper type. In Section 5, we present the formalization of $\mathsf{F}^\blacklozenge_{<:}$, which combines $\lambda^\blacklozenge$ with $\mathsf{F}_{<:}$-style polymorphism for bounded type-and-qualifier abstraction.

3.3 Polymorphic Data Types

In Figure 1, we have used escaping pairs to capture shared mutable data. Now we show how parametric data types such as pairs can work in a suitable extension of $\mathsf{F}^\blacklozenge_{<:}$. In Section 8.1, we study the Church-encoding of pairs in core $\mathsf{F}^\blacklozenge_{<:}$ which conforms to the behaviors laid out here, so that the core formalization does not require built-in pairs. Appendix B of the extended version of this paper [Wei et al. 2023] further studies typing rules for other common data types, *e.g.*, lists.

Suppose that we have extended the language with native pair types, let us first examine how the user-level typing for pairs would work. First, a pair type `Pair[A^a, B^b]` annotates qualifiers to components. Moreover, the projection functions should preserve precise reachability whenever possible. For example, given an expression of type `Pair[A^a, B^b]`, retrieving its components should yield exactly the same qualifiers we put in:

Syntax $\lambda^{\blacklozenge}$

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

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

Fig. 3. The syntax of $\lambda^{\blacklozenge}$. The qualifier shorthands allow us to compactly write unions or singleton sets.

```
...                 // u: Ref[Int]ᵘ, v: Ref[Int]ᵛ          fst(p)    // : Ref[Int]ᵘ
val p = Pair(u, v) //  : Pair[Ref[Int]ᵘ, Ref[Int]ᵛ]ᵖ      snd(p)    // : Ref[Int]ᵛ
```

The above snippet creates a pair of two reference cells and then extracts its components. Type applications are omitted and can be inferred, *e.g.*, by bidirectional typing [Pierce and Turner 2000].

Second, pairs capturing local variables can escape from their defining scope (*e.g.*, the counter example in Figure 1). To this end, we designate a self-reference `p` for pairs μ`p.Pair[A`$^{\mathtt{a}}$`, B`$^{\mathtt{b}}$`]`, which serves as an upper bound of the pair's component reachability. To handle escaped pairs, the key insight is similar to function types: a component qualifier in covariant positions can be upcast to the self-reference of the pair, *i.e.*, `Pair[A`$^{\mathtt{a}}$`, B`$^{\mathtt{b}}$`]` <: μ`p.Pair[A`$^{\mathtt{p}}$`, B`$^{\mathtt{p}}$`]`, just as with function subtyping where the codomain's qualifier can be upcast to the function's self-reference.

```
def f() = { ...     // u: Ref[Int]ᵘ, v: Ref[Int]ᵛ
  Pair(u, v)        //  : Pair[Ref[Int]ᵘ, Ref[Int]ᵛ]{u,v}
}                   //    upcast to μp.Pair[Ref[Int]ᵖ, Ref[Int]ᵖ]♦ when escaping
```

Once the pair is bound to a variable, we "unpack" the self-reference so that projections are properly aliased.

```
// now u and v are not in the context:                    fst(q)    // : Ref[Int]ᵠ
val q = f() // : Pair[Ref[Int]ᵠ, Ref[Int]ᵠ]ᵠ             snd(q)    // : Ref[Int]ᵠ
```

In our Church-encoding of pairs (Section 8.1), the "upcasting with self-reference" is achieved by coercions via η-expansion, which justifies the subtyping rule if added with first-class pair types.

4 SIMPLY-TYPED REACHABILITY POLYMORPHISM

This section presents the formal metatheory of the base $\lambda^{\blacklozenge}$-calculus (Section 3.1), a generalization of the λ^{*}-calculus by Bao et al. [2021] that adds the notion of freshness markers for a more precise notion of lightweight qualifier polymorphism.

4.1 Syntax

Figure 3 shows the syntax of $\lambda^{\blacklozenge}$ which is based on the simply-typed λ-calculus with mutable references and subtyping. We denote general term variables by the meta variables x, y, z, and reserve f, g, h specifically for function self-references in contexts where the distinction matters.

Terms consist of constants of base types, variables, recursive functions $\lambda f(x).t$ (binding the self-reference f and the argument x in the body t), function applications, reference allocations, dereferences, and assignments. Reachability qualifiers p, q, r are finite sets of variables that may additionally include the distinct freshness marker $\blacklozenge$. Once we add store typings (Section 4.3),

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

$$\frac{x : T^{q} \in \Gamma \qquad x \in \varphi}{\Gamma^{\varphi} \vdash x : T^{x}} \text{ (T-VAR)} \qquad \frac{c \in B}{\Gamma^{\varphi} \vdash c : B^{\varnothing}} \text{ (T-CST)}$$

$$\frac{(\Gamma,\ f : F,\ x : P)^{q,x,f} \vdash t : Q \quad q \subseteq \varphi \quad F = (f(x : P) \to Q)^{q}}{\Gamma^{\varphi} \vdash \lambda f(x).t : F} \text{ (T-ABS)}$$

$$\frac{\Gamma^{\varphi} \vdash t : T^{q} \qquad \blacklozenge \notin q}{\Gamma^{\varphi} \vdash \text{ref } t : (\text{Ref } T^{q})^{\blacklozenge q}} \text{ (T-REF)}$$

$$\frac{\Gamma^{\varphi} \vdash t : (\text{Ref } T^{p})^{q} \qquad \blacklozenge \notin p \qquad p \subseteq \varphi}{\Gamma^{\varphi} \vdash !t : T^{p}} \text{ (T-DEREF)}$$

$$\frac{\Gamma^{\varphi} \vdash t_1 : (f(x : T^{p}) \to Q)^{q} \qquad \Gamma^{\varphi} \vdash t_2 : T^{p} \quad \blacklozenge \notin p \qquad Q = U^{r} \qquad r \subseteq \blacklozenge\varphi, x, f \qquad f \notin \text{fv}(U)}{\Gamma^{\varphi} \vdash t_1\ t_2 : Q[p/x, q/f]} \text{ (T-APP)}$$

$$\frac{\Gamma^{\varphi} \vdash t_1 : (f(x : T^{p \cap q}) \to Q)^{q} \qquad \Gamma^{\varphi} \vdash t_2 : T^{p} \quad Q = U^{r} \qquad r \subseteq \blacklozenge\varphi, x, f \quad \blacklozenge \in p \Rightarrow x \notin \text{fv}(U) \quad f \notin \text{fv}(U)}{\Gamma^{\varphi} \vdash t_1\ t_2 : Q[p/x, q/f]} \text{ (T-APP}\blacklozenge\text{)}$$

$$\frac{\Gamma^{\varphi} \vdash t_1 : (\text{Ref } T^{p})^{q} \qquad \Gamma^{\varphi} \vdash t_2 : T^{p} \quad \blacklozenge \notin p}{\Gamma^{\varphi} \vdash t_1 := t_2 : \text{Unit}^{\varnothing}} \text{ (T-ASSGN)}$$

$$\frac{\Gamma^{\varphi} \vdash t : Q \qquad \Gamma \vdash Q <: T^{q} \qquad q \subseteq \blacklozenge\varphi}{\Gamma^{\varphi} \vdash t : T^{q}} \text{ (T-SUB)}$$

Fig. 4. Typing rules of $\lambda^{\blacklozenge}$.

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

$$\frac{}{\Gamma \vdash B <: B} \text{ (S-BASE)} \qquad \frac{\Gamma \vdash P <: O \qquad \Gamma,\ f : (f(x : O) \to Q)^{\blacklozenge},\ x : P \vdash Q <: R}{\Gamma \vdash f(x : O) \to Q <: f(x : P) \to R} \text{ (S-FUN)}$$

$$\frac{\Gamma \vdash S <: T \quad \Gamma \vdash T <: S \quad \Gamma \vdash q_1 <: q_2 \quad \Gamma \vdash q_2 <: q_1}{\Gamma \vdash \text{Ref } S^{q_1} <: \text{Ref } T^{q_2}} \text{ (S-REF)} \qquad \frac{\Gamma \vdash T <: S \quad \Gamma \vdash S <: U}{\Gamma \vdash T <: U} \text{ (S-TRANS)}$$

$$\frac{p \subseteq q \subseteq \blacklozenge dom(\Gamma)}{\Gamma \vdash p <: q} \text{ (Q-SUB)} \qquad \frac{f : T^{q} \in \Gamma \quad \blacklozenge \notin q}{\Gamma \vdash q, f <: f} \text{ (Q-SELF)} \qquad \frac{\Gamma \vdash p <: q \quad \Gamma \vdash q <: r}{\Gamma \vdash p <: r} \text{ (Q-TRANS)}$$

$$\frac{\Gamma \vdash q_1 <: q_2}{\Gamma \vdash p, q_1 <: p, q_2} \text{ (Q-CONG)} \qquad \frac{x : T^{q} \in \Gamma \quad \blacklozenge \notin q}{\Gamma \vdash x <: q} \text{ (Q-VAR)} \qquad \frac{\Gamma \vdash S <: T \qquad \Gamma \vdash p <: q}{\Gamma \vdash S^{p} <: T^{q}} \text{ (SQ-SUB)}$$

Fig. 5. Subtyping rules of $\lambda^{\blacklozenge}$.

qualifiers will include store locations in addition to variables. For readability, we often drop the set notation for qualifiers and write them down as comma-separated lists of atoms.

We distinguish ordinary types T from qualified types $Q = T^{q}$, where the latter annotates a qualifier q to an ordinary type T. The types consist of base types B (*e.g.*, Int, Unit), references, and dependent function types $f(x : P) \to Q$, where both argument and return type are qualified. The codomain Q may depend on both the self-reference f and argument x in its qualifier and type. We could alternatively separate self-references from function types using DOT-style first-class self types [Rompf and Amin 2016], *e.g.*, $\mu f.((x : P) \to Q[f, x])$.

Mutable reference types Ref Q track the known aliases of the value held by the reference. We also permit forms of nested references, which are prohibited in the base λ^{*}-calculus unless a flow-sensitive effect system is added [Bao et al. 2021].

Qualifier Substitution and Growth $\boxed{q[p/x]}$ $\boxed{q[p/\blacklozenge]}$

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

Reachability and Overlap $\boxed{\Gamma \vdash x \rightsquigarrow x}$ $\boxed{\Gamma \vdash q*}$ $\boxed{\Gamma \vdash p \pitchfork q}$

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 $\Gamma \vdash p \pitchfork q := \blacklozenge(p* \cap q*)$

Fig. 6. Operators on qualifiers. We often leave the context Γ implicit (marked as gray).

An *observation* φ is a finite set of variables which is part of the term typing judgment (Section 4.2). It specifies which variables in the static environment Γ are observable. The latter assigns qualified typing assumptions to variables.

4.2 Static Semantics

The term typing judgment $\Gamma^{\varphi} \vdash t : Q$ in Figure 4 states that term t has qualified type Q and may only access the typing assumptions of Γ observable by φ. For $Q = T^{q}$, one may think of t as a computation that yields a result value of type T reaching no more than the transitive closure of q, if it terminates. Alternatively, we could formulate the typing judgment without internalizing φ, and instead have an explicit context filter operation $\Gamma^{\varphi} := \{x : T^{q} \in \Gamma \mid q, x \subseteq \varphi\}$ for restricting the context in subterms, just like Bao et al. [2021] who loosely take inspiration from substructural type systems. Internalizing φ (1) makes observability an explicit notion, which facilitates reasoning about separation and overlap, and (2) greatly simplifies the Coq mechanization. Context filtering is only needed for term typing, but not for subtyping, so as to keep the formalization simple.

4.2.1 One-Step Reachability. Term typing usually assigns *minimal* qualifiers in the currently observable context. For instance, term variables x track exactly themselves (T-VAR), and can be used only if they are observable ($x \in \varphi$). Similarly, constants of base types are untracked (T-CST). We can further scale up the qualifier to include transitively reachable variables by subsumption (T-SUB) if needed. This "one-step" treatment of reachability is sufficient for soundness, and shows that most of the time, we do not have to track fully transitive reachability, but instead may compute it on-demand where it matters, *i.e.*, when checking separation and overlap in function applications (discussed further below). In contrast, Bao et al. [2021] implicitly ensures fully transitive reachability, *i.e.*, term typing always assigns transitively closed qualifiers.[1] Their (T-VAR) rule would assign $T^{q,x}$ where q is transitively closed. One-step reachability simplifies the system and adds finer-grained precision over transitive reachability, since we can refine each step in a reachability chain as more information is discovered during evaluation. Dependent function application and abstraction with function self-references are prime examples (Section 4.2.4).

4.2.2 Functions and Lightweight Polymorphism. Function typing (T-ABS) implements the observable separation guarantee (cf. Section 3.1.2), *i.e.*, the body t can only observe what the function type's qualifier q specifies, plus the argument x and self-reference f, and is otherwise oblivious to anything else in the environment. We model this by setting the observation to q, x, f when typing the body. Thus, its observation q at least includes the free variables of the function. To ensure well-scopedness, q must be a subset of the observation φ. In essence, a function type *implicitly* quantifies over anything that is not observed by q, achieving a lightweight form of qualifier polymorphism.

[1] Cf. their mechanization of this variant https://github.com/tiarkrompf/reachability/tree/main/base/lambda_star_overlap.

4.2.3 Qualifier Substitution and Growth. The base substitution operation $q[p/x]$ of qualifiers for variables is defined in Figure 6, and we use it along with its homomorphic extension to types in dependent function application. Substitution replaces the variable with the given qualifier, if present in the target. We suggestively overload the substitution notation for *qualifier growth* $q[p/\blacklozenge]$. Capturing the intuition behind the freshness marker $\blacklozenge$, growth adds p to q only if $\blacklozenge$ is present, and otherwise ignores p. Growth abstracts over reduction steps that may allocate new reachable store locations in type preservation (Theorem 4.7). We do not remove $\blacklozenge$ to permit continuous growth.

4.2.4 Dependent Application, Separation and Overlap. Function applications are typeable by rules (T-APP) and (T-APP$\blacklozenge$). The former rule applies if the function's parameter is non-fresh ($\blacklozenge \notin p$) and it matches the argument, *i.e.*, the argument qualifier reaches only bound variables and will not increase at run time. Applications in (T-APP) are dependent, substituting the function and argument variable in the type and qualifier of the codomain with the given qualifiers (see Section 3.1.6). Note that the dependency to the argument is unrestricted in the codomain, but the function self-reference can only occur in the outer most return qualifier.

Rule (T-APP$\blacklozenge$) applies to cases where the argument's qualifier is bigger than what the function type assumes, or is expected to grow bigger due to the freshness marker $\blacklozenge$. These cases require more nuanced treatment and restrictions on the degree of dependency in the codomain. That is, if the argument is fresh, then the codomain's type U may not be dependent on the respective variable. Otherwise, type preservation is lost due to the potential growth with fresh runtime locations. In other words, (T-APP$\blacklozenge$) is a synthesis of two specialized rules: If the argument is not fresh, we permit unconstrained dependency to the argument in the codomain:

$$\frac{\Gamma \vdash t_1 : \left(f(x : T^{\,p \pitchfork q}) \to U^{\,r}\right)^{q} \qquad \Gamma \vdash t_2 : T^{\,p} \qquad \blacklozenge \notin p \qquad f \notin \mathrm{fv}(U)}{\Gamma \vdash t_1\ t_2 : U^{\,r}[p/x, q/f]} \quad \text{(T-DAPP)}$$

If the argument is fresh, we require that neither x nor f occur freely the codomain type U (as in Bao et al.).

$$\frac{\Gamma \vdash t_1 : \left(f(x : T^{\,p \pitchfork q}) \to U^{\,r}\right)^{q} \qquad \Gamma \vdash t_2 : T^{\,\blacklozenge p} \qquad \{x, f\} \cap \mathrm{fv}(U) = \varnothing}{\Gamma \vdash t_1\ t_2 : U^{\,r[\blacklozenge p/x, q/f]}} \quad \text{(T-NDAPP)}$$

In all instances of (T-APP$\blacklozenge$), since p is potentially bigger than the function domain, we need to check for *observable separation/overlap* between function and argument, *i.e.*, the portion of p that the function can observe should conform with the function parameter. This is the only place in the type system requiring fully reflexive-transitive reachability using the *overlap operator* $p \pitchfork q$ (Figure 6), which is the intersection of the smallest saturated reachability sets of p and q, always including $\blacklozenge$ to indicate that the argument is allowed to have a bigger qualifier than the domain. For the type safety proof, it is also sufficient to just demand any saturated supersets, as done in our present Coq mechanization.

Both function application rules impose an observability restriction on the codomain qualifier $r \subseteq \blacklozenge\varphi, x, f$, which is to ensure that the resulting qualifier of term typings is always observable under φ (Lemma 4.1), a critical property for the substitution lemmas and type soundness proof.

4.2.5 Mutable References. The λ^* system by Bao et al. [2021] cannot express nested references without the addition of a flow-sensitive effect system. Although extending it with an effect system is possible, our type system readily supports a limited form of nested references by means of reachability and the fresh/non-fresh distinction. Qualifiers in reference types need to be non-fresh in (T-REF), (T-DEREF), and (T-ASSGN). On the outside, reference allocations (T-REF) track the referent's non-fresh qualifier and $\blacklozenge$, because the final result will be a fresh new store location, which will be

added to the qualifier. Since referent qualifiers are invariant, only values with equivalent qualifiers can ever be assigned in (T-ASSGN). Invariance is also reflected in the subtyping rule for references, discussed next.

4.2.6 Subtyping. We distinguish subtyping between qualifiers q, ordinary types T, and qualified types Q, where the latter two are mutually dependent. Subtyping is assumed to be well-scoped under the typing context Γ, *i.e.*, types and qualifiers mention only variables bound in Γ, and so do its typing assumptions. Qualified subtyping (SQ-SUB) just forwards to the other two judgments for scaling the type and qualifier, respectively.

Qualifier Subtyping. Qualifier subtyping includes the subset relation (Q-SUB), the two contextual rules (Q-SELF) and (Q-VAR), and transitivity (Q-TRANS). Rule (Q-SELF) is inherited from Bao et al. [2021], and used for abstracting the qualifiers of escaping closures (Figure 1), *i.e.*, if a function self reference f and its assumed qualifier q occur in some qualifier context, then we may delete q and just retain f, because q may contain captured variables that are not visible in an outer scope. Rule (Q-VAR) is new here and critical for one-step reachability: a qualifier p, x is more precise than p, q since substitution may replace x with a smaller qualifier than q later (cf. Section 3.1.3). This is only valid if $\blacklozenge \notin q$, because otherwise, x could be replaced later with a larger set than q and we would lose track of it. The same restriction applies to (Q-SELF).

Ordinary Subtyping. Subtyping rules for base types (S-BASE), reference types (S-REF), and function types (S-FUN) are standard modulo qualifiers. Reflexivity is admissible for subtyping on ordinary and qualified types. Altough transitivity (S-TRANS) is also admissible, our Coq mechanization relies on invertible value typing as a proof method [Rompf and Amin 2016] that proves subtyping transitivity only for inhabited types. References are invariant both in the enclosed qualifier and the value type, expressed by bidirectional subtype constraints. Function types are contravariant in the domain, and covariant in the codomain, as usual. Due to dependency in the codomain, we are careful to extend the context with the smaller argument type and self reference. Importantly, the function self-reference added to the context only carries the $\blacklozenge$ marker. This distinguishes self-references introduced by term typing in (T-ABS) from synthetic ones for subtyping. Only the former is eligible for abstraction by function self-references, and the freshness marker prevents the use of both (Q-SELF) and (Q-VAR) in the latter case.

4.3 Dynamic Semantics and Stores

The $\lambda^{\blacklozenge}$-calculus adopts the standard call-by-value reduction of the λ-calculus with mutable references and a store (Figure 7). The β-reduction rule simultaneously substitutes the argument with the parameter and the function with the self-reference.

Term typing and subtyping change accordingly to include store typings Σ, and both qualifiers and observations may now include store locations from $\mathrm{dom}(\Sigma)$. Typing a location value (T-LOC) requires that it be observable, along with the full qualifier of the referent ($q, \ell \subseteq \varphi$). This model implements the fully transitive reachability notion for store locations instead of one-step reachability (in contrast to variables, Section 4.2.1), as we never substitute store locations and thus do not alter the assumed qualifiers in the store typing Σ. The well-formedness predicate Σ ok ensures that all assumptions in Σ are closed and have transitively closed qualifiers consisting only of other store locations. Well-formedness is required by Corollary 4.8 to ensure fully disjoint reachability chains and object graphs. The notion of store typing with filter $[\Sigma \mid \Gamma]^{\varphi} \vdash \sigma$ is defined in Figure 7.

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

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

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

Location Reachability, Location & Qualifier Saturation $\boxed{\Gamma \mid \Sigma \vdash \ell \rightsquigarrow \ell}$ $\boxed{\Gamma \mid \Sigma \vdash \ell*}$ $\boxed{\Gamma \mid \Sigma \vdash q*}$

$$\Gamma \mid \Sigma \vdash \ell \rightsquigarrow \ell' \Leftrightarrow \Sigma(\ell) = T^{\,q,\ell'} \qquad \Gamma \mid \Sigma \vdash \ell* := \{\ell' \mid \ell \rightsquigarrow^* \ell'\} \qquad \Gamma \mid \Sigma \vdash q* := \bigcup_{x \in q} x* \cup \bigcup_{\ell \in q} \ell*$$

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

$$\begin{array}{lcl} [\Gamma \mid \Sigma]^{\varphi} \vdash \sigma & := & \varphi \subseteq \mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\Sigma) \wedge \forall \ell \in \varphi, [\Gamma \mid \Sigma]^{\varphi} \vdash \sigma(\ell) : \Sigma(\ell) \\ \Gamma \mid \Sigma \vdash \sigma & := & [\Gamma \mid \Sigma]^{\mathrm{dom}(\Sigma)} \vdash \sigma \end{array}$$

$$\frac{}{\varnothing\ \mathsf{ok}} \qquad \frac{\Sigma\ \mathsf{ok} \quad \mathrm{fv}(T) = \varnothing \quad \mathrm{ftv}(T) = \varnothing \quad \varnothing \mid \Sigma \vdash q* = q \quad \ell \notin \mathrm{dom}(\Sigma)}{\Sigma, \ell : T^{\,q}\ \mathsf{ok}}$$

Reduction Contexts, Values, Terms, Stores

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

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

$$\begin{array}{rcll} C[\,(\lambda f(x).t)\ v\,] \mid \sigma & \longrightarrow & C[\,t[v/x, (\lambda f(x).t)/f]\,] \mid \sigma & (\beta) \\ C[\,\mathsf{ref}\ v\,] \mid \sigma & \longrightarrow & C[\,\ell\,] \mid (\sigma, \ell \mapsto v) \quad \ell \notin \mathrm{dom}(\sigma) & (\textsc{ref}) \\ C[\,!\ell\,] \mid \sigma & \longrightarrow & C[\,\sigma(\ell)\,] \mid \sigma \quad \ell \in \mathrm{dom}(\sigma) & (\textsc{deref}) \\ C[\,\ell := v\,] \mid \sigma & \longrightarrow & C[\,\mathsf{unit}\,] \mid \sigma[\ell \mapsto v] \quad \ell \in \mathrm{dom}(\sigma) & (\textsc{assign}) \\ C[\,(\Lambda f(X^{\,x}).t)\ Q\,] \mid \sigma & \longrightarrow & C[\,t[Q/X^{\,x}, (\Lambda f(X^{\,x}).t)/f]\,] \mid \sigma & (\beta_T) \end{array}$$

Fig. 7. Extension with store typings and call-by-value reduction for $\lambda^{\blacklozenge}$ (Section 4) and $\mathsf{F}^{\blacklozenge}_{<:}$ (Section 5).

4.4 Metatheory

The $\lambda^{\blacklozenge}$-calculus exhibits syntactic type soundness which we prove by standard progress and preservation properties (Theorems 4.6 and 4.7). Type soundness implies the preservation of separation corollary (Corollary 4.8) as set forth by Bao et al. [2021] for their λ^*-calculus. It is a memory property certifying that the results of well-typed $\lambda^{\blacklozenge}$ terms with disjoint qualifiers indeed never alias. We also prove a preservation of separation corollary (Corollary 4.9) for two parallel reductions on disjoint store fragments. Below, we discuss key lemmas required for the type soundness proof, which has been mechanized in Coq. Due to space limitations, we elide standard properties such as weakening and narrowing.

4.4.1 Observability Properties. Reasoning about substitutions and their interaction with overlap/separation in preservation lemmas requires that the qualifiers assigned by term typing are observable. The following lemmas are proved by induction over the respective typing derivations:

Lemma 4.1 (Observability Invariant). *Term typing always assigns observable qualifiers,* i.e.*, if* $[\Gamma \mid \Sigma]^{\varphi} \vdash t : T^{\,q}$*, then* $q \subseteq \blacklozenge\varphi$.

Well-typed values cannot observe anything about the context beyond their assigned qualifier:

Lemma 4.2 (Tight Observability for Values). *If* $[\Gamma \mid \Sigma]^{\varphi} \vdash v : T^{\,q}$*, then* $[\Gamma \mid \Sigma]^{q} \vdash v : T^{\,q}$.

It is easy to see that any observation for a function $\lambda f(x).t$ will at least track the free variables of the body t. Finally, well-typed values are always non-fresh in the following sense:

Lemma 4.3 (Values are Non-Fresh). *If* $[\Gamma \mid \Sigma]^{\varphi} \vdash v : T^{q}$, *then* $[\Gamma \mid \Sigma]^{\varphi} \vdash v : T^{q \setminus \blacklozenge}$.

This lemma is important for substitution, and asserts that values only reach statically fully known variables and locations in context. That is, we may safely assume that values are never the source of $\blacklozenge$, and it can only stem from subsumption, which we may undo by Lemma 4.3. Ruling out $\blacklozenge$ for values ensures that we do not accidentally add it when it is expected to be absent in a substitution target q. The absence indicates that a substitution on q will not increase it with fresh locations.

4.4.2 Substitution Lemma. We consider type soundness for closed terms and apply "top-level" substitutions, *i.e.*, substituting closed values with qualifiers that do not contain term variables, but only store locations. The proof of the substitution lemma critically relies on the distributivity of substitution and the overlap operator (Figure 6), which is required to proceed in the (T-APP$\blacklozenge$) case:

Lemma 4.4 (Top-Level Substitutions Distribute with Overlap). *If* $x : T^{q} \in \Gamma$, *and* $p, q \subseteq \blacklozenge dom(\Sigma)$, *and* $p \cap \blacklozenge\varphi \subseteq q$, *and* $r*, r'* \subseteq \blacklozenge\varphi$, *then* $(r \owedge r')\theta = r\theta \owedge r'\theta$ *where* $\theta = [p/x]$.

Qualifier substitution does not generally distribute with set intersection, due to the problematic case when the substituted variable x occurs in only one of the saturated sets $r*$ and $r'*$. Distributivity holds if (1) we ensure that what is observed about the qualifier p we substitute for x is bounded by what the context observes about x, *i.e.*, $p \cap \blacklozenge\varphi \subseteq q$ for $x : T^{q} \in \Gamma$, and (2) p, q are top-level as above.

In the type preservation proof, β-reduction substitutes both the function parameter and self-reference in (T-ABS) (Figure 4) for some values. The two substitutions can be expressed by sequentially applying a substitution lemma on the first variable in the context:

Lemma 4.5 (Top-Level Term Substitution). *If* $[x : T^{q}, \Gamma \mid \Sigma]^{\varphi} \vdash t : Q$, *and* $[\varnothing \mid \Sigma]^{p} \vdash v : T^{p}$, *and* $p, q \subseteq \blacklozenge dom(\Sigma)$, *and* $p \cap \blacklozenge\varphi \subseteq q$, *and* $q = p \vee q = \blacklozenge(p \cap r)$, *then* $[\Gamma\theta \mid \Sigma]^{\varphi\theta} \vdash t[v/x] : Q\theta$ *where* $\theta = [p/x]$.

Proof. By induction over the derivation $[x : T^{q}, \Gamma \mid \Sigma]^{\varphi} \vdash t : Q$. Most cases are straightforward, exploiting that qualifier substitution is monotonic w.r.t. $\subseteq$ and that the substitute p for x consists of store locations only. The case (T-APP$\blacklozenge$) critically requires Lemma 4.4 for $(p \owedge q)\theta = p\theta \owedge q\theta$ in the induction hypothesis. The case (T-SUB) requires an analogous substitution lemma for subtyping (elided due to space limitations). □

Just as above, the substitution lemma imposes the observability condition $p \cap \blacklozenge\varphi \subseteq q$. The condition $q = p \vee q = \blacklozenge(r \cap p)$ captures the two different cases of substitution: (1) a precise substitution where the assumed qualifier q for x is identical to the value's qualifier p, *i.e.*, the parameter in (T-APP) or the function's self-reference f in (T-APP)/(T-APP$\blacklozenge$), or (2) a growing substitution for the parameter in (T-APP$\blacklozenge$) with overlap between p and the function qualifier r, growing the result by $p \setminus r*$.

4.4.3 Main Soundness Result.

Theorem 4.6 (Progress). *If* $[\varnothing \mid \Sigma]^{\varphi} \vdash t : Q$ *and* Σ ok, *then either* t *is a value, or for any store* σ *where* $[\varnothing \mid \Sigma]^{\varphi} \vdash \sigma$, *there exists a term* t' *and store* σ' *such that* $t \mid \sigma \longrightarrow t' \mid \sigma'$.

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

Similar to [Bao et al. 2021], reduction preserves types up to qualifier growth (cf. Section 4.2.3):

Theorem 4.7 (Preservation). *If* $[\varnothing \mid \Sigma]^{\varphi} \vdash t : T^{q}$, *and* $[\varnothing \mid \Sigma]^{\varphi} \vdash \sigma$, *and* $t \mid \sigma \longrightarrow t' \mid \sigma'$, *and* Σ ok, *then there exists* $\Sigma' \supseteq \Sigma$, $\varphi' \supseteq \varphi \cup p$, *and* $p \subseteq \mathrm{dom}(\Sigma' \setminus \Sigma)$ *such that* $[\varnothing \mid \Sigma']^{\varphi'} \vdash \sigma'$ *and* $[\varnothing \mid \Sigma']^{\varphi'} \vdash t' : T^{q[p/\blacklozenge]}$.

Proof. By induction over the derivation $[\varnothing \mid \Sigma]^{\varphi} \vdash t : T^{q}$. □

Corollary 4.8 (Preservation of Separation). *Sequential reduction of two terms with disjoint qualifiers preserve types and disjointness:*

$$\frac{\begin{array}{llll}[\varnothing \mid \Sigma]^{\mathrm{dom}(\Sigma)} \vdash t_1 : T_1^{q_1} & t_1 \mid \sigma \longrightarrow t_1' \mid \sigma' & \varnothing \mid \Sigma \vdash \sigma & \Sigma \text{ ok} \\ [\varnothing \mid \Sigma]^{\mathrm{dom}(\Sigma)} \vdash t_2 : T_2^{q_2} & t_2 \mid \sigma' \longrightarrow t_2' \mid \sigma'' & q_1 \pitchfork q_2 \subseteq \{\blacklozenge\} & \end{array}}{\begin{array}{lll}\exists p_1\ p_2\ \Sigma'\ \Sigma''. \quad [\varnothing \mid \Sigma']^{\mathrm{dom}(\Sigma')} \vdash t_1' : T_1^{p_1} & \Sigma'' \supseteq \Sigma' \supseteq \Sigma \\ [\varnothing \mid \Sigma'']^{\mathrm{dom}(\Sigma'')} \vdash t_2' : T_2^{p_2} & p_1 \pitchfork p_2 \subseteq \{\blacklozenge\}\end{array}}$$

Proof. By sequential application of Preservation (Theorem 4.7) and the fact that a reduction step increases the assigned qualifier by at most a fresh new location, thus preserving disjointness. □

Corollary 4.9 (Progress and Preservation in Parallel Reductions). *Non-value expressions with disjoint observability filters can be evaluated in parallel on non-overlapping parts of the store ($\sigma_{\restriction_\varphi}$ restricts the domain of σ to locations in φ), and the resulting qualifiers remain separate:*

$$\frac{\begin{array}{llll}[\varnothing \mid \Sigma]^{\varphi_1} \vdash t_1 : T_1^{q_1} & [\varnothing \mid \Sigma]^{\varphi_1} \vdash \sigma & t_1, t_2 \textit{ non-value} & \Sigma \text{ ok} \\ [\varnothing \mid \Sigma]^{\varphi_2} \vdash t_2 : T_2^{q_2} & [\varnothing \mid \Sigma]^{\varphi_2} \vdash \sigma & \varphi_1 \cap \varphi_2 \subseteq \varnothing & \end{array}}{\begin{array}{llll}\exists \sigma_1'\ \sigma_2'\ \Sigma_1\ \Sigma_2\ p_1\ p_2\ \varphi_1'\ \varphi_2'. & & & \\ t_1 \mid \sigma_{\restriction_{\varphi_1}} \longrightarrow t_1' \mid \sigma_1' & [\varnothing \mid \Sigma_1]^{\varphi_1'} \vdash t_1' : T_1^{p_1} & \Sigma_1 \supseteq \Sigma & \\ t_2 \mid \sigma_{\restriction_{\varphi_2}} \longrightarrow t_2' \mid \sigma_2' & [\varnothing \mid \Sigma_2]^{\varphi_2'} \vdash t_2' : T_2^{p_2} & \Sigma_2 \supseteq \Sigma & p_1 \pitchfork p_2 \subseteq \{\blacklozenge\}\end{array}}$$

Proof. Since φ_1 and φ_2 are disjoint, by Lemma 4.1, q_1 and q_2 are also disjoint. By Progress (Theorem 4.6), t_1 and t_2 can be reduced to t_1' and t_2', respectively. Then by Preservation (Theorem 4.7), the contractums are well-typed. With disjoint new locations picked for the two reductions, the resulting qualifiers p_1 and p_2 are also disjoint. In a real system, this requires a thread-safe allocator with synchronization between two parallel threads or thread-local allocation pools. □

5 REACHABILITY AND TYPE POLYMORPHISM

We extend the simply-typed reachability-polymorphic system $\lambda^{\blacklozenge}$ with type-and-qualifier abstraction in the style of $\mathsf{F}_{<:}$ [Cardelli et al. 1994]. The typing of this extension behaves the same as in standard $\mathsf{F}_{<:}$ modulo self-references and reachability sets. As mentioned in Section 3.2, we simultaneously abstract over both types and qualifiers.

5.1 Syntax

Figure 8 shows the syntax of $\mathsf{F}^{\blacklozenge}_{<:}$ as a $\mathsf{F}_{<:}$-style extension of $\lambda^{\blacklozenge}$. Types now include the Top type, type variables X, and universal types. A universal type introduces a quantified type variable X along with a quantified qualifier variable x, which are both upper-bounded by a qualified type Q. It is important to read the combined quantification as an abbreviation introducing the abstract type and qualifier independently, as they do not need to be used together, *i.e.*, $\forall(X <: T).\forall(x <: q).Q \equiv \forall(X^{x} <: T^{q}).Q$. We choose the more compact syntax for readability since types and qualifiers are often instantiated together. Similar to function types, universal types have self-references, which are useful when a polymorphic closure escapes its defining scope. The body of a universal type is also qualified and can access the self-reference f of the universal type in addition to x. Terms now include type abstractions and qualified type applications. Type abstractions bind their own self-reference f, type parameter X, and qualifier parameter x in the body t. Typing environments now include bounded type-and-qualifier variables of the form $X^{x} <: Q$.

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

$$
\begin{array}{llll}
T & ::= & \cdots \mid \mathsf{Top} \mid X \mid \forall f(X^{x} <: Q).Q & \text{Types} \\
t & ::= & \cdots \mid \Lambda f(X^{x}).t \mid t\ [Q] & \text{Terms} \\
\Gamma & ::= & \cdots \mid \Gamma, X^{x} <: Q & \text{Typing Environments}
\end{array}
$$

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

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

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

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

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

$$\frac{X^{x} <: T^{q} \in \Gamma \qquad \blacklozenge \notin q}{\Gamma \vdash p, x <: p, q} \quad (\textsc{q-qvar}) \qquad \frac{}{\Gamma \vdash T <: \mathsf{Top}} \quad (\textsc{s-top})$$

$$\frac{X^{x} <: T^{q} \in \Gamma}{\Gamma \vdash X <: T} \quad (\textsc{s-tvar}) \qquad \frac{\Gamma \vdash Q <: O \qquad \Gamma\ ,\ f : (\forall f(X^{x} <: O).P)^{\blacklozenge}\ ,\ X^{x} <: Q \vdash P <: R}{\Gamma \vdash \forall f(X^{x} <: O).P <: \forall f(X^{x} <: Q).R} \quad (\textsc{s-all})$$

Fig. 8. The syntax and typing rules of $\mathsf{F}^{\blacklozenge}_{<:}$ as an extension of $\lambda^{\blacklozenge}$.

5.2 Static Semantics

The typing and subtyping rules of $\mathsf{F}^{\blacklozenge}_{<:}$ (Figure 8) are a superset of those presented for $\lambda^{\blacklozenge}$ in Section 4.

5.2.1 Typing Rules. We add the typing rules for type abstractions and type applications. The type system is defined declaratively in Curry-style, and hence for type abstractions (T-TABS) we need to "guess" the whole universal type and its qualifier. Other parts are analogous to term abstraction typing (Section 4.2.2). Notably, observable separation naturally generalizes to type abstraction. That is, the qualifier q constrains what the type abstraction's implementation can observe, and P's qualifier in $X^{x} <: P$ determines observable overlap/separation for instantiations of x. Especially, if P mentions the freshness marker $\blacklozenge$, then instantiations of x can mention unobserved variables.

Similar to function applications in $\lambda^{\blacklozenge}$, there are two type application rules: (T-TAPP) for non-fresh dependent applications and (T-TAPP$\blacklozenge$) for restricted dependent applications. Requiring non-freshness ensures that we pass a qualifier argument that is bounded by other variables in the context. Rule (T-TAPP$\blacklozenge$) is analogous to (T-APP$\blacklozenge$) (cf. Section 4.2.4): If the argument qualifier is fresh, then the result type U cannot be dependent on it. We impose observability constraints on the codomain qualifier r to ensure the observability invariant (Lemma 4.1) for $\mathsf{F}^{\blacklozenge}_{<:}$.

5.2.2 Subtyping Rules. Rule (S-TOP) is the standard rule for the Top type. Instead of defining a single subtyping rule for type-and-qualifier variables that simply looks up the context in the premise, we disentangle it to the (Q-QVAR) rule and (S-TVAR) rule, distinguishing the subtyping for qualifiers and ordinary types (cf. Section 4.2.6). The former accounts for subtyping of qualifiers, allowing upcasting a qualifier variable to its upper bound. The latter is akin to standard type variable subtyping. This

disentanglement reflects the fact that we can upcast the quantified qualifier and type independently, despite that they are introduced together using a combined syntax.

For universal types (S-ALL), we use the "full" subtyping rule for richer expressiveness [Curien and Ghelli 1992] where type bounds are contravariant. This rule renders subtyping an undecidable relation [Pierce 1992] (see Section 6 for discussion on decidability). Due to self-references, we also extend the context with the smaller universal type when subtyping the body, as in DOT [Rompf and Amin 2016]. Note that (S-ALL) invokes subtyping on qualified types in its premises. Similar to the base system, typing transitivity in $\mathsf{F}^{\blacklozenge}_{<:}$ can be admissible, but our Coq mechanization relies on it along with invertible value typing [Rompf and Amin 2016].

5.3 Dynamic Semantics and Metatheory

Figure 7 highlights the changes and new rules of $\mathsf{F}^{\blacklozenge}_{<:}$'s dynamic semantics, as an extension of $\lambda^{\blacklozenge}$. The reduction semantics is entirely standard compared to $\mathsf{F}_{<:}$, the only difference being that type abstractions are recursive, so that type application (β_T) also substitutes the type abstraction itself along with the argument. Since location typing (T-LOC) and store well-formedness require closed types in store typings, we additionally demand the absence of free type variables.

$\mathsf{F}^{\blacklozenge}_{<:}$ enjoys the same soundness properties as $\lambda^{\blacklozenge}$, *i.e.*, progress, preservation, and the separation of preservation corollary (cf. Section 4.4.3). As for $\lambda^{\blacklozenge}$, we have proved these results in Coq for $\mathsf{F}^{\blacklozenge}_{<:}$.

6 IMPLEMENTATION CONSIDERATIONS

Accompanying the declarative type systems and their metatheory, in this section we discuss our experience in implementing a prototype type checker[2] for the $\mathsf{F}^{\blacklozenge}_{<:}$-calculus. The type checker accepts an enriched input language (similar to the surface language used in Section 2) with qualified types and `val`/`def` definitions in addition to the core calculus.

Bidirectional Typing and Annotations. We adopt the recipes of bidirectional typing [Dunfield and Krishnaswami 2022] so that the prototype supports both type checking and type inference. As in Scala, users still need to annotate function argument types and qualifiers, but can omit most annotations on `val`-bound variables, function returns, and type applications. Some syntactic sugar is provided, *e.g.*, untracked types do not need to be explicitly annotated. More precise user-provided annotations are still accepted and checked against the inferred types by subtyping.

Decidability. The $\lambda^{\blacklozenge}$-calculus is based on the simply-typed λ-calculus with subtyping and references, which is known to be decidable. Our addition, *i.e.*, qualifiers in $\lambda^{\blacklozenge}$ (including their transitive closures), are finite sets, and therefore are not a source of undecidability, either. The polymorphic $\mathsf{F}^{\blacklozenge}_{<:}$-calculus is developed on top of the "full" variant of $\mathsf{F}_{<:}$, which is known to be undecidable due to the subtyping of universal types [Pierce 1992]. However, the choice of using the "full" variant is orthogonal to our calculus and our focus of this paper is type soundness and preservation of separation. In practice, undecidable subtyping is not a severe concern. Many languages such as Rust or Scala do not have decidable typing, yet are still practically useful. In our case, it is possible to obtain a decidable fragment by building atop the "kernel" variant of $\mathsf{F}_{<:}$.

Qualifier Subtyping and Inference. At the core of our prototype is the *algorithmic* version of subtype checking, which also decides the subqualifier relation. Similar to the "exposure" algorithm [Pierce 2002] for type variables in $\mathsf{F}_{<:}$, checking sub-qualifier relations induces (1) exposing the qualifiers to their largest equivalent following Q-SELF (cf. Figure 5), and (2) checking their subsumption recursively following Q-VAR.

[2]The prototype can be found at https://github.com/tiarkrompf/reachability.

Additionally, our prototype supports inferring supertypes with self-references in many cases. This is the key to make function applications with fresh arguments work without user annotations, where deep dependency of bound variables is disallowed when the argument is fresh (cf. T-APP♦ in Figure 4). Such deep dependencies must be upcast to self-references before application. An example is escaping pairs that have been presented in Section 3.3:

```
def f() = { ...     // u: Ref[Int]^u, v: Ref[Int]^v
  Pair(u, v)        //  : Pair[Ref[Int]^u, Ref[Int]^v]^{u,v}
}                   //  upcast to μp.Pair[Ref[Int]^p, Ref[Int]^p]♦ when escaping
```

To avoid inferring ill-typed qualifiers, our prototype detects escaping names (*e.g.*, u and v) and upcasts them to the self-reference p by Q-SELF, as they occur in covariant positions. The full formalization and metatheory of inference are topics beyond the scope of this paper and will be addressed in future work.

From Prototype to Scala. Scaling the prototype to a full-blown language is a long way to go, but an interesting question to ask. Scala 3 now has an experimental implementation [Odersky et al. 2023] for capture types [Boruch-Gruszecki et al. 2023], which provides the basic mechanism of tracking captured variables in functions in addition to existing Scala concepts such as self-references and bidirectional type inference. It should be thus possible to implement our proposal of reachability types on top of the infrastructure of capture types. To achieve that, reachability types have to be extended to handle Scala's rich notions of types and objects. Also, constraint resolution of qualifiers is required to integrate with Scala's type inference [Odersky et al. 2001]. Despite the substantial engineering effort, bringing reachability types to Scala would be beneficial to introduce the freshness notion and separation guarantee, enabling new applications such as safe parallelism in Scala. In Section 8.2, we compare the expressiveness of capture types and our proposal.

7 LIMITATIONS & EXTENSIONS

In this section, we discuss a few directions to extend the base type system for richer expressiveness.

7.1 Nested Mutable References

Section 4.2.5 has presented the formalization of nested references. This is already an expressiveness improvement compared to Bao et al. [2021], which requires layering a flow-sensitive effect system on top the base system. However, our nested references are also limited in supporting escaping or cyclic references. Here we discuss possible extensions to address these limitations.

Nested References in $\mathsf{F}^{\blacklozenge}_{<:}$. With nested references, a reference's content also carries a reachability annotation, *e.g.*, $\texttt{Ref[T}^{\texttt{p}}\texttt{]}^{\texttt{q}}$, where T can be any type as long as p is not fresh. That is, only references with fully observable reachability are permitted, and these references remain invariant once introduced, and can only be assigned with values having the same reachability set.

This pattern permits more flexible uses of capabilities, *e.g.*, registering effectful functions as callbacks or tracking permissible escaping via assignments. Recall the counter example (Figure 1) that returns two functions to increase or decrease an encapsulated state. Both functions share the same reachable set containing ctr. Note that both functions encapsulate and mutate a locally-defined heap reference cell, thus are effectful. $\mathsf{F}^{\blacklozenge}_{<:}$ allows to create a reference cell that stores either the fst(ctr) or snd(ctr) function:

```
val ctr = counter(0)       // : Pair[(()=>Unit)^ctr,(()=>Unit)^ctr]^ctr
val cf = new Ref(fst(ctr)) // : Ref[(()=>Unit)^ctr]^cf
cf := snd(ctr)             // : Unit^∅
```

Escaping Nested References. In the current system, although it is possible to store a reference cell into another, the outer reference cell cannot escape to a scope that does not observe the inner reference. Consider the following example that attempts to return a nested reference:

```
def f(n: Int) = {
  val c1 = Ref(n)  // : Ref[Int]^c1
  val c2 = Ref(c1) // : Ref[Ref[Int]^c1]^c2
  c2               // cannot return as either Ref[Ref[Int]^∅]♦ or Ref[Ref[Int]♦]♦
}
```

At the end of function `f`, we cannot properly type `c2` due to its inner qualifier `c1`. Since `c1` is not visible once `f` returns, we have to replace it with something else. However, assigning the untracked empty set or the freshness marker would violate our type soundness guarantee (*i.e.*, losing tracking status or aliasing). Part of the difficulty is because reference types do not have self-references as function types, so that we cannot use the self-reference (of the outer reference cell) to encapsulate or upcast the inner qualifier. Yet, it is still possible to escape `c2` by eta-expanding the reference with a pair of fractional capabilities to *read* or *write* the reference cell [Boyland 2003; Reynolds 1988], so that we can make use of the self-references of these functions:

```
def f(n: Int) = {
  ... // same as before
  (() => !c2, (m: Ref[Int]^c1) => c2 := m) // : Pair[(() => Int)^c2, (Ref[Int]^c1 => ())^c2]
} // upcast to μp.Pair[(() => Int)^p, (Ref[Int]^∅ => ())^p]
```

It is important to note that after packing the internal qualifier with the pair's self-reference, the write capability's argument qualifier has been narrowed from `{c1}` to ∅, since it is in a *contravariant* position. Thus, the write capability is not applicable once escaped, since there is no untracked reference in our system. Still, this paradigm can offer a useful fractional capability to write functions or other maybe-tracked data, and generally when the argument qualifier can be narrowed to a non-empty set.

Cyclic References. Another limitation in the current system is that it disallows cyclic references, so we cannot create data structures such as doubly-linked lists or graphs. All these limitations suggest that we should also add *self-references* μ`p.Ref[T^q]` to reference types. This would enable expressing cyclic references with type μ`p.Ref[T^p]`.

However, this can only be sound with a careful treatment of the self-reference. It is known that monolithic reference types are invariant, but we can allow upcasting reachability to its enclosing self-reference only at *covariant* (as in pairs) but not *contravariant* positions. Therefore, to properly address the limitation of escaping or cyclic references with self-references, we would need to refine references to two fractional qualifiers or types, and deploy different self-reference rules for them (see the above example).

7.2 Move Semantics and Uniqueness via Flow-Sensitive Effects

In Section 2, we have demonstrated several examples enabled by extending the base reachability type system with a flow-sensitive effect system. This is useful to track fine-grained behaviors such as move semantics, ownership transfer, deallocation, etc. Bao et al. [2021] have presented a fruitful marriage of effects and reachability by augmenting types with effect tracking. This section briefly discusses how to adopt Bao et al.'s flow-sensitive effect system to $\mathsf{F}^{\blacklozenge}_{<:}$.

Applying the effect system extension of Bao et al. to $\mathsf{F}^{\blacklozenge}_{<:}$ gives rise to the following augmented type, reachability, and effect judgement $\Gamma^{\varphi} \vdash t : T^{q} \mid \epsilon$, where ϵ is an *alias-aware effect store* that maps sets of variables to their effects. For example, the following store instance records the fact that aliased variables x and y induce a read effect and z induces a write effect: $\langle \{x, y\} \mapsto \mathsf{rd}, \{z\} \mapsto \mathsf{wr} \rangle$.

Informed by the qualifiers, possibly overlapping components and their effects are combined while checking effects. The alias-aware effect store is an instance of Gordon [2021]'s generic sequential effect system using effect quantales; see Bao et al. for more details.

The effect extension of Bao et al. relies on obtaining transitively reachable sets when building up the alias-aware effect store. This is necessary to ensure any overlapping check is sound. However, as discussed in Section 4.2.1, our system $\mathsf{F}^{\blacklozenge}_{<:}$ assigns minimal one-step reachability when typing terms. Luckily, in $\mathsf{F}^{\blacklozenge}_{<:}$ we can still request saturated reachable sets when necessary, which is already used in the (T-APP♦) rule (Figure 4). In fact, we could just use the $q*$ defined in Figure 6 to obtain the fully saturated qualifier. We illustrate this idea with the type-and-effect version of reference assignment (E-ASSIGN) (other rules can be extended similarly):

$$\frac{\Gamma^{\varphi} \vdash t_1 : (\mathsf{Ref}\ T^{p})^{q} \mid \epsilon_1 \quad \Gamma^{\varphi} \vdash t_2 : T^{p} \mid \epsilon_2 \quad \blacklozenge \notin p}{\Gamma^{\varphi} \vdash t_1 := t_2 : \mathsf{Unit}^{\varnothing} \mid \epsilon_1 \rhd \epsilon_2 \rhd \langle q \mapsto \mathsf{wr} \rangle} \ (\text{E-ASSIGN})$$

The effects of subterm t_1 and t_2 are ϵ_1 and ϵ_2, respectively. The interesting part is that an assignment term has its own intrinsic effect, which imposes a write effect wr over all aliases of q. These three sub-effects are sequentially composed using $\rhd$, which merges overlapped domains and their effects w.r.t. their execution order. The definitions of the sequential composition operator $\rhd$ and the alias-aware effect store follows the one presented by Bao et al. [2021], additionally computing transitive closures when necessary. In $\mathsf{F}^{\blacklozenge}_{<:}$, it is possible that q is the singleton set of the freshness marker ♦. In this case, the effect composition yields that the empty set is associated with some effect, indicating some *non-observable* effectful operation has happened.

Linearity, Affinity, and Uniqueness. It has been folklore that linearity and uniqueness are dual to each other, and their formal relation have been carefully revisited recently [Marshall et al. 2022]. Our effect system extension is closer to uniqueness and ownership types, which guarantees a variable uniquely holds the resource via move semantics. With the "kill" effect, we can express *affinity* that an entity is killed once it is used, *i.e.*, used at most once (see examples in Section 2.2). With a possible "must-reachable" extension of reachability types, we can also implement the notion of "at least used once", which leads to linearity when combined with "at most used once".

Consumption vs Use & Mention. Traditional substructural type systems use the concept of "consumption" to check linearity or affinity (*e.g.* [Bernardy et al. 2018]). However the common notion of "consumption" fails to distinguish the "use" and "mention" of a resource [Gordon 2020]. The former is considered effectful and the later can be considered pure. This is partly due to the lack of effect information. In this work, with a proper integration of alias tracking and effect systems, the purity of a term becomes evident, allowing us to safely mention a resource (*e.g.*, a pointer) multiple times, but enforcing effectful uses at most once.

8 CASE STUDIES

8.1 Church-Encodings of Polymorphic Data Structures

Section 3.3 has explained how polymorphic pairs can behave in a suitable extension of $\mathsf{F}^{\blacklozenge}_{<:}$. Their behavior is justifiable by Church-encodings in the core calculus. We distinguish between "transparent" pairs and "opaque" pairs. Transparent pairs track precise reachability of components using $\mathsf{F}^{\blacklozenge}_{<:}$'s parametric qualifiers and can only be used under appropriate contexts. Opaque pairs use self-references as an abstraction to hide local qualifiers and can escape to an outer scope. Finally, we justify the connection between transparent and opaque pairs via subtyping, since there is a coercion function that eta-expands pairs, converting transparent pairs to opaque pairs.

Typing Church Pairs, Transparently. The transparent pair type `Pair[A, B]` is defined as a universal type with argument type `C`. We also introduce abstract qualifiers along with types `A`, `B`, and `C`. The qualifier of result type `C` is simply parametric.

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

We also assume the base system is extended with multi-argument functions (instead of currying arguments), where each argument is disjoint from others. Similarly, the term constructor uses `C`'s qualifier for the application `f(a, b)`:

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

When using the quantified type for the argument or return type, its accompanying qualifier is implicitly attached, *i.e.*, we write `A` as a shorthand of A^a when using it.

The projectors `fst` and `snd` have their usual definitions but using accurate types and qualifiers:

```
def fst[A^a <: Top^♦, B^b <: Top^{a, ♦}](p: Pair[A^a, B^b]^{a,b,♦}): A^a = p((a, b) => a)
def snd[A^a <: Top^♦, B^b <: Top^{a, ♦}](p: Pair[A^a, B^b]^{a,b,♦}): B^b = p((a, b) => b)
```

By making the elimination type `C`'s qualifier parametric, we can instantiate it in the projection function with the precise component qualifiers, as shown by the example at the beginning of Section 3.3.

Typing Escaped Church Pairs, Opaquely. The transparent pair typing works for cases where the components are still in the context, but the pair cannot escape from that scope (cf. Figure 1). We now discuss the types of escaped pairs using *self-references* as abstraction. To avoid confusion, we name the type and constructor of opaque pairs as `OPair`, and transparent pairs remain `Pair`.

```
type μp.OPair[A^a <: Top^♦, B^b <: Top^{a,♦}] = // p: self-reference of a pair instance
  p[C^c <: Top^∅] => (h((x: A^♦, y: B^{x,♦}) => C^{x,y}) => C^h)^p
```

Recall that in $F^{\blacklozenge}_{<:}$ universal types and type abstractions also have self-references (*e.g.*, `p` in the definition) that can be used to express escaping polymorphic closures, similar to their term-level correspondences (*e.g.*, `h` in the definition). Therefore, the self-reference in μ`p.OPair` is just a syntactic annotation referring to the self-reference of the universal type. Compared to the transparent typing, here we do not use quantified qualifiers that are parametrically introduced. Instead, we use a chain of self-references in the codomains, upcasting from the inner most reachability `{x, y}` to `h` and to `p`. The introduction and elimination forms of opaque pairs also reflect the typing using self-references:

```
def OPair[A^a <: Top^♦, B^b <: Top^{a,♦}](a: A, b: B): μp.OPair[A, B]^{a,b} =
  [C^c <: Top^∅] => (f: (x: A^♦, y: B^{x,♦}) => C^{x,y}) => f(a, b)
def fst[A^a <: Top^♦, B^b <: Top^{a,♦}](p: μp.OPair[A, B]^{a,b}): A^p = p((a, b) => a)
def snd[A^a <: Top^♦, B^b <: Top^{a,♦}](p: μp.OPair[A, B]^{a,b}): B^p = p((a, b) => b)
```

Imprecise Eliminations. While the typing works out, the resulting qualifiers of the projections `fst`/`snd` are imprecise. We have no means to vary the qualifier of the elimination type `C` in type `OPair`. When the component qualifiers are not available in the context, using the self-reference to track possible sharing is the most accurate option. This is the intended design as discussed in Section 3.3. A side effect of such a typing is that in-scope elimination can yield the set of joint qualifiers, since the pair can reach them by our "maybe-tracked" notation:

```
...                 // u and v defined as before
val p = OPair(u, v) // : μp.OPair[Ref[Int], Ref[Int]]^♦ binds to p, unpacking the self-ref
fst(p)              // : Ref[Int]^p <: Ref[Int]^{u,v} ← imprecise joint qualifiers
snd(p)              // : Ref[Int]^p <: Ref[Int]^{u,v} ← imprecise joint qualifiers
```

Conversion between Opaque and Transparent Pairs. The two different types for Church-encoded pairs are connected, *i.e.*, transparent pairs can be converted to opaque via eta-expansion:

```
def conv[A^a <: Top♦, B^b <: Top^{a,♦}](p: Pair[A, B]^{a,b,♦}): μp.OPair[A, B]^{a,b} =
  OPair(fst(p), snd(p))
```

From a pragmatic perspective, when the language is extended with pairs as native algebraic data types, the eta-expansion justifies an admissible subtyping rule for escaped pairs.

General Data Types. The general Church-encoding of data types via sums and products can also benefit from the increased precision. In Appendix B [Wei et al. 2023], we discuss the typing rules of other generic data types, including boxes, options, and lists.

8.2 Comparison with Scala Capture Types

A closely related work to ours is the recent Scala capture types (CT) proposal [Boruch-Gruszecki et al. 2023] which also tracks sets of variables. The system is tailored to programming with effects as non-escaping capabilities, providing a lightweight form of effect polymorphism. In this section, we inspect a few aspects of CT and demonstrate its limitation in handling fresh resources.

Capture Sets, Universal Qualifier, and Box Types. Similar to our system, capture types are built on top of $\mathsf{F}_{<:}$ and types can be annotated with variable sets, *i.e.*, $\{c_1, \ldots, c_n\}\ T$ where c_i is a variable representing the captured capability. Here is a (simplified) combinator for scoped exception handling using capture types [Odersky et al. 2021]:

```
// declares the throw capability:
class CanThrow
// passes a tracked non-escaping capability to a block:
def _try[A](block: (c: {*} CanThrow) -> A) = block(CanThrow())
```

Importantly, `{*}` is a special marker for the top element for qualifier subtyping in capture types, meaning some unknown set of variables is tracked, *e.g.*, `{*} CanThrow` above.

While superficially similar, this top qualifier should not be confused with our ♦ marker indicating a fresh/growing qualifier, and behaves differently, as we will show later.

Since `c` represents a universal capability, we want to enforce that the lifetime of capability `c` passed to the given `block` is bound to the scope of `_try`. In other words, it should not be leaked for any given `block`, *e.g.*, by directly returning it or returning it indirectly through an escaping closure. Capture types enforce this by requiring that the universal capability `{*}` cannot escape. This is in contrast to capabilities bound to a variable in an outer scope.

When combined with parametric type polymorphism, Boruch-Gruszecki et al. [2023] propose to use a box type operator $\square\ T$ to turn qualified types into proper, unqualified types, so that type variables only need to range over proper types. A boxed value $\square[q\ T]$ capturing local variables in q is upcast to $\square[\{*\}\ T]$ when going out of scope. Unboxing such types recovers the capture set. Boxed values can only be unboxed if the contained qualifier q is a concrete variable set, specifically excluding the top qualifier `{*}`. This provides a mechanism for statically enforcing non-escaping capabilities, *i.e.*, boxes are implicitly inserted at the abstraction boundary whenever the block's return type `A` is instantiated with a tracked type:

```
// illegal use (escaping capabilities):
val x = _try { c => c }       //: □[{*} CanThrow], error: cannot box/unbox
val y = _try { c => () => c } //: □[{*} (() -> CanThrow)], error : cannot box/unbox
```

On the outside, subtyping can only assign the `{*}` qualifier to blocks that return or capture the capability `c`, since the captured variable is not visible in the outer scope.

Limitation: Tracking Fresh Values. Let us now consider combining `_try` with other resources that have non-scoped introduction forms and should be tracked:

```
// assume freshAlloc() : {*} T
val outer = freshAlloc()          // : {*} T is bound to {outer} T
val z = _try { c => () => outer } // : □[{outer} (() -> T)], ok: can box/unbox
```

The compiler rejects unboxing the `{*}` qualifier, but allows it for any more concrete one. However, while the box type prevents capabilities from escaping, the compiler must infer and insert box introductions and eliminations at declaration and use sites of polymorphic terms. But more importantly, the capture type mechanism does not support unbound fresh values well, *e.g.*, fresh allocations. The obvious choice is assigning the top-qualifier `{*}` to indicate some new value, but this is at odds with boxing/unboxing, *e.g.*, one cannot write

```
val fresh  = _try { c => freshAlloc() }                 // : □[{*} T], error
val fresh2 = _try { c => val f = freshAlloc(); () => f } // : □[{*} (() -> T)], error
```

A potential workaround is having a separate global capability (*e.g.*, `heap`) for allocations:

```
// fresh3 : {fresh2} T <: {heap} T
val fresh3 = _try { c => heap.freshAlloc() } // : □[{heap} T], ok: can unbox
```

This solution works well for effects-as-capabilities models, but it is unsatisfactory for tracking aliasing and separation, *e.g.*, all fresh values have a common super type `{heap} T` which pollutes subtyping chains and leads to a loss of distinction between separate fresh allocations. In summary, if we want to track the lifetimes of a given class of resources, these lifetimes must be properly nested in a stack-like manner with the lifetimes of all other resources.

The Reachability Approach. Our $\mathsf{F}^{\blacklozenge}_{<:}$-calculus can correctly handle fresh values, while at the same time not requiring a box type. This stems from (1) having an intersection operator for reasoning about separation/overlap, and (2) a strong observability guarantee on function types and universal types. For instance, here is the type- and qualifier-polymorphic version of `_try`:

```
// ∀A^z <: Top^♦. ((CanThrow^♦ → A^{z,♦})^♦ → A^{z,♦})
def _try[A^♦](block: ((c: CanThrow^♦) => A)^♦): A = block(CanThrow())
```

The annotation on the `block` parameter specifies that it is contextually fresh for the implementation of `_try` and thus entirely separate in terms of transitive reachability. We still reject the `x` and `y` examples above (Section 8.2), but we now permit `fresh: T^♦` and `fresh2: f(() => T^{f})`, which correctly preserves the freshness of unnamed results. Finally, $\mathsf{F}^{\blacklozenge}_{<:}$ permits finer-grained type distinctions when returning fresh values, due to function self references:

```
// CT:  {*} (() -> T)                  // CT:  {*} (() -> T)
// F♦<:: () => T^♦                     // F♦<:: f(() => T^{f})
def retFresh() = () => freshAlloc()    def retConst() = { val f = freshAlloc(); () => f }
```

$\mathsf{F}^{\blacklozenge}_{<:}$ distinguishes between returning a fresh value on each invocation, versus returning one and the same fresh value escaping a local scope, whereas both are indistinguishable in capture types.

Outlook. Compared to CT, reachability types exhibit similar expressiveness and can support all relevant uses of capture types. Additionally, reachability types show richer expressiveness in a few key aspects, especially the tracking of freshness and the guarantee of separation. In future work, we propose to implement reachability types on top of the experimental CT implementation [Odersky et al. 2023] in Scala 3, which would provide additional guarantees, such as separation. We look forward to seeing how these two lines of similar ideas can benefit each other in the future.

9 RELATED WORK

Tracking Variables in Types. The most directly related work of this paper is the original work on reachability types [Bao et al. 2021]. This paper addresses key limitations of Bao et al. [2021] and improves its expressiveness by introducing a new reachability tracking mechanism, the freshness marker, and type-and-qualifier quantification.

Capture types [Boruch-Gruszecki et al. 2021; Boruch-Gruszecki et al. 2023; Odersky et al. 2021, 2022] is another recent ongoing effort to integrate capability tracking and escaping checking into Scala 3. Several calculi have been proposed for capture types, *e.g.*, $\mathsf{CF}_{<:}$ [Boruch-Gruszecki et al. 2021] and $\mathsf{CC}_{<:\Box}$ [Odersky et al. 2021, 2022]. In Section 8.2, we have discussed and compared with capture types. To achieve capture tunnelling with universal polymorphism, the $\mathsf{CC}_{<:\Box}$ calculus uses boxing/unboxing, inspired by contextual modal type theory (CMTT) [Nanevski et al. 2008]. Scherer and Hoffmann [2013] propose open closure types used for data-flow analysis where function types carry their defining lexical environments. Several type systems [Jang et al. 2022; Kiselyov et al. 2016; Parreaux et al. 2018] designed for manipulating open code in metaprogramming also track free variables and contexts in types, which are closely related to CMTT.

Escaping, Freshness, and Existential Types. Works inspired by regions [Tofte and Talpin 1997] use existential types for tracking freshness or escaping entities, *e.g.*, in Alias types [Smith et al. 2000], L^3 [Ahmed et al. 2007], and Cyclone [Grossman et al. 2002], analogous to our freshness marker and self-reference. As an analogy, one can think of a type with the freshness marker $\texttt{Ref}^{\blacklozenge}$ as having an underlying quasi-existential type $\mu x.\texttt{Ref}^{\{x\}}$ where the reference type tracks its own self-reference. However, existentials for this purpose in our system would have to preserve precise reachability information across temporary aliases created during pack/unpack operations. That is, special facilities simulating the freshness marker and related constructs would need to be used in the implementation of existentials, if those were taken as primitives. Therefore, we believe the typing with self-references is more concise and appropriate than existentials here, because we can use the *same* variable. In addition, the use of self-references for escaping closures in our work makes reasoning about them more succinct. Similar to our calculi, type systems distinguishing second-class values can also enforce non-escaping properties of effects or capabilities [Brachthäuser et al. 2022, 2020; Osvald et al. 2016; Siek et al. 2012; Xhebraj et al. 2022]. To regain the ability to return second-class capabilities, Brachthäuser et al. [2022] again make use of boxing and unboxing.

Separation. The notion of separation and the intersection operator (Section 4.2.4) used in reachability types is inspired by separation logic [O'Hearn et al. 2001; Reynolds 2002] and its predecessors [O'Hearn et al. 1999; Reynolds 1978, 1989]. Bunched typing [O'Hearn 2003] and syntactic control of interference [O'Hearn et al. 1999; Reynolds 1978, 1989] allow reasoning about disjoint and shared resource access. This is similar to reachability types, however, our system does not enforce that the *computations* of the function and arguments are disjoint, but their final *values* are disjoint (rule T-APP$\blacklozenge$ in Figure 4). Bunched typing enforces separation by splitting the typing context, whereas our work enforces separation by checking disjointness of *saturated* reachability sets. Bunched typing also lacks an explicit treatment of aliasing.

Uniqueness types [Barendsen and Smetsers 1996; de Vries et al. 2006, 2007] ensure that there is no more than one reference pointing to the resource, effectively establishing separation. Marshall et al. [2022] present a language unifying linearity [Wadler 1990] and uniqueness. Our base system does not directly track either linearity or uniqueness, instead, flow-sensitive "kill" effects that disable all aliases can be integrated to statically enforce uniqueness [Bao et al. 2021].

Polymorphism. Reachability types and our variants feature lightweight reachability polymorphism without introducing explicit quantification (cf. Section 4.2.2). Capture types [Boruch-Gruszecki

et al. 2021; Odersky et al. 2021, 2022] provide a similar flavor via dependent function application. Brachthäuser et al. [2022, 2020] propose to represent effects as capabilities, which yields a lightweight form of effect polymorphism that requires little annotations.

Various forms of polymorphism exist in prior work on ownership types. Noble et al. [1998] use generic parameters to pass aliasing modes into a class. But they do not allow ownership parameterization isolated from type parameterization. Clarke [2003] further supports ownership polymorphism via context parameters. Similarly, Ownership Generic Java [Potanin et al. 2006] allows programmers to specify ownership information through type parameters. Jo∃ [Cameron and Drossopoulou 2009; Cameron 2009] combines the theory of existential types with a parametric ownership type system, where ownership information is passed as additional type arguments. Generic Universe Types [Dietl et al. 2011] integrate the owners-as-modifiers discipline with type genericity, effectively separating the ownership topology from the encapsulation constraints.

Collinson et al. [2008] combine F-style polymorphism with bunched logic, where universal types are discerned to be either additive and multiplicative, but do not allow abstraction over additivity and multiplicativity. Our system $\mathsf{F}^{\blacklozenge}_{<:}$ has quantified abstraction over qualifiers, which can be used as an argument's reachability, permitting flexible instantiations of either disjointness or sharing.

Constraints in alias types [Smith et al. 2000] support a form of location and store polymorphism, where the latter abstracts over irrelevant store locations. Our calculi implicitly abstract over contexts by baking the observability notion into typing.

Ownership Types. Ownership type systems [Clarke et al. 1998; Noble et al. 1998] are generally concerned with objects in OO programs and start from the uniqueness restriction [Boyapati et al. 2002; Clarke et al. 2001; Dietl et al. 2011; Müller and Poetzsch-Heffter 2000; Zhao et al. 2008] and then selectively re-introduce sharing in a controlled manner [Clebsch et al. 2015; Hogg 1991; Naden et al. 2012]. Inherited from Bao et al. [2021], our calculi are designed for higher-order languages and deem sharing and separation as essential substrates, on top of which an additional effect system can be layered to achieve uniqueness and ownership transfer. The focus of this paper is to address the limitations in expressiveness of Bao et al. [2021] regarding reachability and type polymorphism.

Rust's type system [Matsakis and Klock 2014] enforces strict uniqueness of mutable references, while immutable references can be shared via borrowing, known as the "shared XOR mutable" rule. Mezzo [Balabonski et al. 2016] is a language designed for controlling aliasing and mutation, and share some similarities with $\mathsf{F}^{\blacklozenge}_{<:}$. Mezzo tracks aliasing using singleton types [Smith et al. 2000]. When dealing with effects, Mezzo imposes restrictions like Rust: mutable portions of the heap must have a unique owner, whereas reachability types relax this constraint. Moreover, Mezzo lacks the notion of separation between functions and arguments and uses existential quantification to handle escaping functions that capture local variables. $\mathsf{F}^{\blacklozenge}_{<:}$ checks separation at the call site and has a lightweight mechanism to track escaping functions via self-references.

Typestate-oriented programming [Aldrich et al. 2009] and its combination with gradual typing [Garcia et al. 2014] also provides static flow-sensitive reasoning or dynamic enforcement.

10 CONCLUSION

In this work, we propose a new reachability type system $\lambda^{\blacklozenge}$ that has lightweight, precise, and sound reachability polymorphism. Based on $\lambda^{\blacklozenge}$, we add bounded quantification over types and qualifiers, leading to a type-and-reachability-polymorphic calculus $\mathsf{F}^{\blacklozenge}_{<:}$. We have formalized these systems and proved the soundness and separation guarantees in Coq. We further discuss implementation considerations and possible directions to extend the base system for richer expressiveness. We compare our system with Scala capture types in the context of programming with capabilities. Our system subsumes both prior reachability types and the essence of Scala capture types, while exhibiting richer expressiveness in key aspects such as modeling freshness and guaranteeing separation.

ACKNOWLEDGMENTS

We thank Siyuan He and Haotian Deng for related contributions to reachability types, and the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by NSF awards 1553471, 1564207, 1918483, 1910216, DOE award DE-SC0018050, as well as gifts from Meta, Google, Microsoft, and VMware.

REFERENCES

Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L^3: A Linear Language with Locations. *Fundam. Informaticae* 77, 4 (2007), 397–449.

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *OOPSLA Companion*. ACM, 1015–1022.

Roberto M. Amadio and Luca Cardelli. 1993. Subtyping Recursive Types. *ACM Trans. Program. Lang. Syst.* 15, 4 (1993), 575–631.

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.

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The Design and Formalization of Mezzo, a Permission-Based Programming Language. *ACM Trans. Program. Lang. Syst.* 38, 4 (2016), 14:1–14:94.

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.

Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Math. Struct. Comput. Sci.* 6, 6 (1996), 579–612.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* 2, POPL (2018), 5:1–5:29.

Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, Ondrej Lhoták, and Martin Odersky. 2021. Tracking Captured Variables in Types. *CoRR* abs/2105.11896 (2021).

Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.* (sep 2023). Just Accepted.

Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. 2002. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*. ACM, 211–230.

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 55–72.

Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 1–30.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 126:1–126:30.

Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 236 (oct 2023), 31 pages.

Nicholas Cameron and Sophia Drossopoulou. 2009. Existential Quantification for Variant Ownership. In *ESOP (Lecture Notes in Computer Science, Vol. 5502)*. Springer, 128–142.

Nicholas Robert Cameron. 2009. *Existential Types for Variance - Java Wildcards and Ownership Types*. Ph. D. Dissertation. Imperial College London, UK.

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1994. An Extension of System F with Subtyping. *Inf. Comput.* 109, 1/2 (1994), 4–56.

David Clarke. 2003. *Object Ownership and Containment*. Ph. D. Dissertation. University of New South Wales.

David Clarke, James Noble, and John Potter. 2001. Simple Ownership Types for Object Containment. In *ECOOP (Lecture Notes in Computer Science, Vol. 2072)*. Springer, 53–76.

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

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

Sylvan Clebsch, Sebastian Blessing, Juliana Franco, and Sophia Drossopoulou. 2015. Ownership and reference counting based garbage collection in the actor world. In *ICOOOLPS'2015*. ACM.

Matthew Collinson, David J. Pym, and Edmund Robinson. 2008. Bunched polymorphism. *Math. Struct. Comput. Sci.* 18, 6 (2008), 1091–1132.

Pierre-Louis Curien and Giorgio Ghelli. 1992. Coherence of Subsumption, Minimum Typing and Type-Checking in $F_{\leq}$. *Math. Struct. Comput. Sci.* 2, 1 (1992), 55–91.

Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. 2006. Uniqueness Typing Redefined. In *IFL (Lecture Notes in Computer Science, Vol. 4449)*. Springer, 181–198.

Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. 2007. Uniqueness Typing Simplified. In *IFL (Lecture Notes in Computer Science, Vol. 5083)*. Springer, 201–218.

Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2011. Separating ownership topology and encapsulation with generic universe types. *ACM Trans. Program. Lang. Syst.* 33, 6 (2011), 20:1–20:62.

Jana Dunfield and Neel Krishnaswami. 2022. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2022), 98:1–98:38.

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 36, 4 (2014), 12:1–12:44.

Colin S. Gordon. 2020. Designing with Static Capabilities and Effects: Use, Mention, and Invariants (Pearl). In *ECOOP (LIPIcs, Vol. 166)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:25.

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

Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *PLDI*. ACM, 282–293.

John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. In *OOPSLA*. ACM, 271–285.

Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Mœbius: metaprogramming using contextual types: the stage where system f can pattern match on itself. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–27.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34.

Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers - Type- and Scope-Safe Code Generation with Mutable Cells. In *APLAS (Lecture Notes in Computer Science, Vol. 10017)*. 271–291.

Daniel Marshall, Michael Vollmer, and Dominic Orchard. 2022. Linearity and Uniqueness: An Entente Cordiale. In *ESOP (Lecture Notes in Computer Science, Vol. 13240)*. Springer, 346–375.

Nicholas D. Matsakis and Felix S. II Klock. 2014. The Rust language. In *HILT*. ACM, 103–104.

Peter Müller and Arnd Poetzsch-Heffter. 2000. A type system for controlling representation exposure in Java. In *ECOOP Workshop on Formal Techniques for Java Programs*.

Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. 2012. A type system for borrowing permissions. In *POPL*. ACM, 557–570.

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

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

Martin Odersky et al. 2023. *Scala 3 Reference - Capture Checking*. https://docs.scala-lang.org/scala3/reference/experimental/cc.html

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

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

Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored local type inference. In *POPL*. ACM, 41–53.

Peter W. O'Hearn. 2003. On bunched typing. *J. Funct. Program.* 13, 4 (2003), 747–796.

Peter W. O'Hearn, John Power, Makoto Takeyama, and Robert D. Tennent. 1999. Syntactic Control of Interference Revisited. *Theor. Comput. Sci.* 228, 1-2 (1999), 211–252.

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.

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.

Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2018. Unifying analytic and statically-typed quasiquotes. *Proc. ACM Program. Lang.* 2, POPL (2018), 13:1–13:33.

Benjamin C. Pierce. 1992. Bounded Quantification is Undecidable. In *POPL*. ACM Press, 305–315.

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44.

Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. 2006. Generic ownership for generic Java. In *OOPSLA*. ACM, 311–324.

John C. Reynolds. 1978. Syntactic Control of Interference. In *POPL*. ACM Press, 39–46.

John C. Reynolds. 1988. Preliminary design of the programming language Forsythe. *Tech Report, CMU-CS-88-159, Carnegie Mellon University* (1988).

John C. Reynolds. 1989. Syntactic Control of Inference, Part 2. In *ICALP (Lecture Notes in Computer Science, Vol. 372)*. Springer, 704–722.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74.

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *OOPSLA*. ACM, 624–641.

Gabriel Scherer and Jan Hoffmann. 2013. Tracking Data-Flow with Open Closure Types. In *LPAR (Lecture Notes in Computer Science, Vol. 8312)*. Springer, 710–726.

Jeremy G. Siek, Michael M. Vitousek, and Jonathan D. Turner. 2012. Effects for Funargs. *CoRR* abs/1201.0023 (2012).

Frederick Smith, David Walker, and J. Gregory Morrisett. 2000. Alias Types. In *ESOP (Lecture Notes in Computer Science, Vol. 1782)*. Springer, 366–381.

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

Philip Wadler. 1990. Linear Types can Change the World!. In *Programming Concepts and Methods*. North-Holland, 561.

Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2023. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs (Extended Version). *CoRR* abs/2307.13844 (2023).

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.

Tian Zhao, Jason Baker, James Hunt, James Noble, and Jan Vitek. 2008. Implicit ownership types for memory management. *Science of Computer Programming* 71, 3 (2008), 213–241.

Litao Zhou, Yaoda Zhou, and Bruno C. d. S. Oliveira. 2023. Recursive Subtyping for All. *Proc. ACM Program. Lang.* 7, POPL (2023), 1396–1425.

Yaoda Zhou, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2022. Revisiting Iso-Recursive Subtyping. *ACM Trans. Program. Lang. Syst.* 44, 4 (2022), 24:1–24:54.

Received 2023-07-11; accepted 2023-11-07