Reachability Types: Tracking Aliasing and Separation in Higher-Order Functional Programs

YUYAN BAO, University of Waterloo, Canada
GUANNAN WEI, Purdue University, USA
OLIVER BRAČEVAC, Purdue University, USA
YUXUAN JIANG, Purdue University, USA
QIYANG HE, Purdue University, USA
TIARK ROMPF, Purdue University, USA

Ownership type systems, based on the idea of enforcing unique access paths, have been primarily focused on objects and top-level classes. However, existing models do not as readily reflect the finer aspects of nested lexical scopes, capturing, or escaping closures in higher-order functional programming patterns, which are increasingly adopted even in mainstream object-oriented languages. We present a new type system, λ^*, which enables expressive ownership-style reasoning across higher-order functions. It tracks sharing and separation through reachability sets, and layers additional mechanisms for selectively enforcing uniqueness on top of it. Based on reachability sets, we extend the type system with an expressive flow-sensitive effect system, which enables flavors of move semantics and ownership transfer. In addition, we present several case studies and extensions, including applications to capabilities for algebraic effects, one-shot continuations, and safe parallelization.

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

Additional Key Words and Phrases: reachability types, ownership types, aliasing, type systems, effect systems

ACM Reference Format:
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, Article 139 (October 2021), 32 pages. https://doi.org/10.1145/3485516

1 INTRODUCTION

Unrestricted aliasing spells trouble because it inhibits local reasoning—for programmers just as well as for compilers and analysis tools. Hence, type systems that control aliasing hold great promise to improve safety, performance, and expressiveness in programming. And after decades of active research, ownership type systems are indeed on the brink of mainstream adoption. With Rust [Matsakis and Klock 2014] as the primary driver, similar models are being explored for Swift [The Swift Developer Community 2019], D [Bright 2019], and other languages, all based on the pioneering academic work on ownership and borrowing [Clarke et al. 1998; Hogg 1991; Noble et al. 1998], region type systems [Grossman et al. 2002; Tofte and Talpin 1997], uniqueness

Authors' addresses: Yuyan Bao, University of Waterloo, Canada, yuyan.bao@uwaterloo.ca; Guannan Wei, Purdue University, USA, guannanwei@purdue.edu; Oliver Bračevac, Purdue University, USA, bracevac@purdue.edu; Yuxuan Jiang, Purdue University, USA, jiang700@purdue.edu; Qiyang He, Purdue University, USA, he615@purdue.edu; Tiark Rompf, Purdue University, USA, tiark@purdue.edu.

2475-1421/2021/10-ART139
https://doi.org/10.1145/3485516

```
def counter(n: Int) = {
  val c = new Ref(n)
  (() => c += 1, () => c -= 1)
}
val (incr, decr) = counter(0)
incr()
decr() // result: 0
```

(a) An idiomatic `counter` implementation.

```
fn counter(n: i64)->(impl Fn()->(), impl Fn()->()) {
   let c = Rc::new(Cell::new(n));
   let c1 = c.clone();
   let c2 = c.clone();
   (move || { c1.set(c1.get() + 1); },
    move || { c2.set(c2.get() - 1); })
}
```

(b) A Rust implementation of `counter` using ref. counting (RC).

Fig. 1. Many programming patterns involving higher-order functions that are common in ML, Scheme, or Scala are not expressible under a Rust-style ownership and borrowing discipline. Here, (a) shows a function `counter` that returns two closures that communicate through a shared captured reference cell. Stepping outside the "shared XOR mutable" paradigm means that Rust implementations cannot rely on static lifetime tracking and need to use mechanisms like dynamic reference counting instead (b).

types [Barendsen and Smetsers 1996], as well as linear and affine type systems [Wadler 1990]. What is more, availability of such type systems has enabled new programming patterns beyond the immediate uses of safe static memory management and optimization. In particular, systems that control lifetimes of values give rise to programming models based on capabilities with context-sensitive lifetime and sharing properties [Haller and Loiko 2016; Steed and Drossopoulou 2016], also leading to new perspectives on effect systems, especially lightweight effect polymorphism [Brachthäuser et al. 2020a; Osvald et al. 2016].

However, even though tremendous progress has been made in making ownership type systems expressive enough for mainstream use, state-of-the-art systems still restrict the use of certain high-level language features in significant ways. The arguably most important restrictions are placed on functional abstraction, *i.e.*, the use of higher-order functions and lexical closures, which have become an indispensable tool even in imperative languages like Java and C++. Programming patterns based on closures that capture and communicate through shared pieces of mutable state, which are common in ML and other impure functional languages, are not readily supported by strict "shared XOR mutable" disciplines as in Rust.[1] As a concrete example, consider the program in Fig. 1, which creates a functional counter abstraction, returning two closures for increment and decrement operations that both close over the same internal mutable reference. While such examples can of course be made to work in Rust, this generally requires stepping outside the static ownership and borrowing model and relying on dynamic reference counting or unsafe operations.

From Ownership and Borrowing to Reachability and Separation. How can we remove such restrictions and enable expressive ownership-style reasoning across higher-order functions? Surprisingly little research exists in this direction, beyond more restrictive models like regions [Tofte and Talpin 1997] or linear types [Bernardy et al. 2018; Wadler 1990]. Most prior work is focused primarily on objects and classes, and while it is certainly true that functions can be implemented as objects, ownership models focused on top-level classes do not reflect the finer aspects of nested lexical scopes, capturing, and escaping closures, as the Rust example shows.

Existing ownership systems generally enforce a topological structure on the heap. In this regard, ownership has been likened to structured programming [Noble 2018]. Many variations exist, from uniqueness types [Barendsen and Smetsers 1996] to region type systems (heap as stack of regions) [Tofte and Talpin 1997], to the original ownership model (heap structured as a tree) [Clarke et al. 1998], to even more flexible topological models [Clarke et al. 2013; Dietl et al. 2011; Müller and Poetzsch-Heffter 2000]. However, to be practical, all such systems that enforce unique access need to be layered with an auxiliary notion such as borrowing [Hogg 1991] that selectively re-introduces sharing, although in highly restricted ways; typically, for the duration of a method call.

[1] In Rust, a mutable reference is unique, and a read-only reference may be shared.

With the goal of expanding the applicability of ownership-style reasoning, it bears asking if global heap invariants are really necessary? In the world of low-level verification of imperative programs, a powerful alternative exists: separation logic [Reynolds 2002] enables reasoning about wild pointer-chasing programs using only few well-placed annotations that demand separation of individual heap references, as a binary predicate satisfied by mutual non-reachability, and without imposing global restrictions on the layout of the heap. However, scaling separation logic to higher-order programs comes with nontrivial challenges [Krishnaswami et al. 2009] and type systems that are close cousins of separation logic [O'Hearn 2003; Reynolds 1978] enforce the stricter notion of complete non-interference of computations, rather than separation of individual heap objects. Nevertheless, forms of higher-order concurrent separation logic [Jung et al. 2018b; Krebbers et al. 2017] have been pivotal in establishing formal foundations for Rust [Jung et al. 2018a, 2021], including semantic interpretions of Rust types. Hence, wouldn't it make sense to build more of this underlying separation substrate into the level of user-facing syntactic types?

From Reachability and Separation back to Ownership-Style Reasoning. In this paper, we investigate the possibility of recasting ownership-style reasoning on a foundation inspired by reachability and separation. In a sense, we turn the prevailing model of alias control via type systems upside down: instead of restricting sharing and then selectively re-introducing it through borrowing, we propose a type system that tracks sharing—and its absence, separation—at its core, and then layer additional mechanisms for selectively enforcing uniqueness on top of it.

We present a *reachability type system*, λ^*, that introduces a type qualifier to track reachability sets in a straightforward extension of simply-typed lambda calculus with mutable references. The reachable set of a function is the combined reachable set of its free variables, consistent with an interpretation of functions as closure records. Function arguments are assumed to be separate from any of the function's free variables, unless declared otherwise. To enforce this invariant, function application requires separation between the function *itself* and the argument. However, tracking the free variables of a function naively would give up many useful properties—in particular the essential closure abstraction that a function can escape its defining scope and continue to interact with its captured environment in meaningful ways, while conforming to a purely extensional signature. We solve this key problem by allowing self-references in function types, a concept borrowed from the DOT (Dependent Object Types) family of type systems [Amin et al. 2016; Rompf and Amin 2016], and rewiring references to free variables in the type of an escaping closure to self-references to the closure itself. This handles use cases like the one in Fig. 1. We prove a type-and-reachability soundness theorem and show as a corollary that reduction preserves separation.

The base λ^* reachability type system is kept intentionally simple and does not permit flow-sensitive reasoning, which would be required to reflect the changing reachability structure of nested layers of mutable state in the form of move semantics, ownership transfer, etc. Hence, mutable references in the base system are restricted to contain immutable data only. Still, the base system is already surprisingly expressive and effective at modeling various sharing policies, including effects-as-capabilities models. While it would be possible to extend the base system directly with flavors of ownership transfer, etc., we pursue a different route: based on the realization that reachability information is a great basis for tracking side effects in a fine-grained way, we present a generic effect system, λ^*_ϵ, based on a novel notion of *store-sensitive* effect quantales [Gordon 2021] which can be instantiated in a variety of flow-insensitive and flow-sensitive ways. As it turns out, all that is needed to support nested mutable state, unique references, move semantics, linearity, and more, is a notion of *kill effects* that disable any further access to a value and its aliases. Specifically, unique acess is achieved by killing all *other* access paths, with many additional practical applications such as use-once capabilities.

In summary, this paper makes the following contributions:

- We introduce the λ^* reachability type system informally, with examples, and with a particular focus on reachability qualifiers and functional abstraction (Section 2).
- We present the formal typing rules as an extension of the simply-typed λ-calculus with mutable reference cells and prove a type-and-reachability safety theorem (Section 3).
- We present a generic effect system λ^*_ϵ and its instantiation with flow-sensitive use and kill effects, which enables nested mutable references with move semantics (Section 4).
- We present several case studies and extensions, including an application to Brachthäuser et al. [2020a]'s capabilities for algebraic effects, control operators for one-shot continuations, and safe parallelization (Section 5).

We discuss related work in Section 6 and offer concluding remarks in Section 7. Prototype implementations and Coq mechanizations of selected λ^* variants are available online at: https://github.com/tiarkrompf/reachability

2 REACHABILITY TYPES: INFORMAL INTRODUCTION

We introduce the λ^* type system and the notion of reachability types informally, using a series of motivating examples. They are presented in Scala-like syntax, but the presented approach is applicable to any impure functional programming language.

2.1 Tracking Reachability with Type Qualifiers

Types in λ^* are designed to track which other values are reachable from a given expression's result. For practicality, we allow certain values to be untracked, such as base types and pure functions, but we ensure that all mutable values are tracked, and by extension all values that may point to them.

```
val x = 4                  // : Int
val y = new Ref(7)         // : Ref[Int]{y}
```

Types in λ^* are of the form T^q, where q is a *reachability qualifier*: either $\bot$, denoting an untracked value, or a set of variable names, denoting the set of tracked values that are reachable. We often omit $\bot$ qualifiers.

In the code snippet above, the type of variable `x` is `Int`, and the type of variable `y` is `Ref[Int]`$^{\{y\}}$, a heap-allocated mutable reference cell. The reachability qualifier of `y` means the variable `y` is aliased with itself. Variable declarations may introduce new aliasing:

```
val z = y                  // : Ref[Int]{y,z}
```

The type of `z` is `Ref[Int]`$^{\{y,z\}}$, which is derived from the type of `y` by adding `z` to the qualifier.

Reachability qualifiers can be applied to any type, not just reference cells, and thus turn values of any type into tracked values. In particular, functions that close over tracked values will become tracked as well (see Section 2.2).

2.1.1 Fresh Values. What is the type of a freshly allocated reference that is *not* bound to a variable? Since it is not reachable in any other way, it has an empty reachability qualifier:

```
new Ref(7)                 // : Ref[Int]∅
```

Such values with the empty set $\emptyset$ as qualifier are called *fresh values*: they are *tracked* but not currently reachable through the environment. Note that the empty set $\emptyset$ as qualifier is different from $\bot$, which signifies *untracked* values. When a fresh value is bound to a variable, that variable will be added to the qualifier. Assigning a tracked type to the only introduction form for mutable references guarantees that all mutable references are tracked.

It is also important to note that "fresh" does not mean globally unique in this context. A fresh value may still be bound in another scope, *e.g.*, in the caller's environment when inside a function.

2.1.2 Subtyping. Reachability qualifiers represent an upper bound on the actual set of reachable values; they should be read as "may-reach" but not "must-reach", although a corresponding extension is possible (see Section 3.9). Using subtyping, untracked values can be treated as tracked, and additional variables can be added to the reachability qualifier any time using subtyping. However, the type system ensures that tracked values can never be treated as untracked.

Subtyping for functions, references, etc., is subject to the usual variance rules.

2.2 Functional Abstraction

Functions can reach all tracked values they close over. Hence, the reachability qualifier of a function is the combined reachable set of all its free variables:

```
val c = ...                          // : Ref[Int]{c,...}
def addInt(x: Int) = { c += x }      // : (Int => Unit){c,...}
addInt(7)                            // : Unit
```

This is consistent with the interpretation of functions as closure records. A function is untracked if it does not close over any tracked values. Functions may have tracked or untracked arguments and return tracked or untracked values, independent of their own tracking status.

2.2.1 Function Arguments. Function arguments are assumed to be separate from tracked variables in the environment, unless declared otherwise. Below, we declare the argument `c` as fresh, with the intention to rule out any troublesome aliasing between `c1` and `c` that might lead to interference:

```
val c1 = ...                         // : Ref[Int]{c1}
val c2 = ...                         // : Ref[Int]{c2}
def addRef(c: Ref[Int]∅) = {         // : (Ref[Int]∅ => Unit){c1}
  c1 += get(c)
}
addRef(c1)                           // error
addRef(c2)                           // ok
```

As expected, calling `addRef(c1)` is a type error. However, disallowing an invocation `addRef(c2)` would seem overly restrictive: since `addRef` has no way to access `c2` it would never be able to *observe* aliasing between `c2` and `c`!

Hence, we settle for guaranteeing *observable separation* by checking that the argument is not aliased with any of the function's free variables. Since the function itself tracks all its free variables, function application just needs to check that the function itself is separate from the argument.

To permit overlap between the argument and any free variables, we have to declare it explicitly:

```
def addRef(c: Ref[Int]{c1}) = ...    // : (Ref[Int]{c1} => Unit){c1}
```

Now the invocation `addRef(c1)` will succeed as well.

2.2.2 Function Return Values. Functions can return tracked values from three different sources: (1) the environment, (2) their own argument, and (3) freshly allocated or obtained through an external call. The three situations are summarized below:

```
// : Ref[Int]{c0}
val c0 = ...
// : (Int => Ref[Int]{c0}){c0}
def returnEnv(x: Int) = { c0 += x; c0 }
// : (x:Ref[Int]∅) => Ref[Int]{x}
def returnArg(x: Ref[Int]∅) = { x += 1; x }
// : Int => Ref[Int]∅
def returnFresh(x: Int) = { val y = new Ref(x); y += 1; y }
```

```
// : Ref[Int]{c1,c0}
val c1 = returnEnv(7)
// : Ref[Int]{c2,c0}
val c2 = returnArg(c0)
// : Ref[Int]{c3}
val c3 = returnFresh(7)
```

There are several points to note: (1) `returnEnv` becomes tracked, (2) `returnArg` has a dependent function type, so that the result is tracked to alias the argument provided at the call site, (3) internal

tracked bindings such as `y` in `returnFresh` that are not visible from the outside are removed from the reachability qualifier of the result.

2.3 Escaping Closures

Let us take a closer look at the typings involved when tracked objects escape the current scope; `returnFresh` above is a simple example. In many cases it is safe and sufficient to simply remove any variables that become unavailable from the reachability qualifier. Below, the arrows indicate how the type assigned inside the block changes to the one assigned externally:

```
val y = new Ref(0); y  // : Ref[Int]{y} ⟶ Ref[Int]∅
```

In this case, removing `y` from the reachability set yields exactly the desired behavior. The external type signals a fresh reference, *i.e.*, it is indistinguishable from that of a reference allocated freshly, without the intermediate binding. In the following example, the reference cell remains reachable from an escaping closure, and we can again just remove `y`. The empty reachability set will make sure that the closure remains tracked:

```
val y = new Ref(0)
() => { y += 1; }      // : (() => Unit){y} ⟶ (() => Unit)∅
```

But if we try some more variations we quickly run into trouble with tracked types occurring in nested positions of the result type. Consider the following example, in which the reference cell can directly leak through an escaping closure:

```
val y = new Ref(0)
() => y                // : (() => Ref[Int]{y}){y} ⟶ (() => Ref[Int]∅)∅ (wrong!)
```

Now we have a problem: each external invocation of the escaped function will be typed as returning a `Ref[Int]`$^{\varnothing}$, which looks like it would be a fresh reference cell. But this is incorrect: there is only one reference cell, returned again by each invocation!

We solve this key problem by allowing self-references in function types, a concept known from Scala and DOT [Amin et al. 2016; Amin and Rompf 2017]. Using subtyping, we rewire references to free variables in the type of an escaping closure to self-references to the closure itself, before eliminating unavailable variables from the *top-level* reachability qualifier only (see Section 3 for details).

```
val y = new Ref(0)
() => y  // : (() => Ref[Int]{y}){y} ⟶ (f() => Ref[Int]{f}){y} ⟶ (f() => Ref[Int]{f})∅
```

Now each such invocation will return a reference cell that visibly aliases the function. Thus, all returned references are shared, which is exactly what we want.

2.4 Key Higher-Order Programming Patterns

2.4.1 Encapsulating Shared State. Functional abstraction combines primitives into implementations of an extensional interface, without exposing the inner workings. Often that involves encapsulating hidden state. Here, we revisit the example from Fig. 1, a counter abstraction that can only be accessed through its increment and decrement operations. The assigned typings are as follows:

```
// : Int=>Pair[p=>(()=>Unit){p},(()=>Unit){p}]∅
def counter(n: Int) = {
  val c = new Ref(n)
  // : Pair[(()=>Unit){c},(()=>Unit){c}]{c}
  (() => c += 1, () => c -= 1)
}
```

```
// : Pair[(()=>Unit){p},(()=>Unit){p}]{p}
val p = counter(0)
// : (()=>Unit){incr,p}
val incr = fst(p)
// : (()=>Unit){decr,p}
val decr = snd(p)
```

Like functions, pairs also need a self-reference `[p => ...]` to track the fact that both returned closures share state. Note that `counter`'s return type has the empty set ∅ as its qualifier. This means that the function returns a fresh, *i.e.*, tracked but unbound, value. At the call side, the value is bound

to p, resulting in qualifier p, which indicates that the value is reachable through p. In our formal model, we encode pairs and other data types as functions (see Section 3.8), but it would also be possible to support this in a regular OO class system that has self types, such as Scala. It is useful to think of the returned pair as "owning" the two closures, however the correct reading is simply that parts of p are reachable from `incr`/`decr`; no ownership or hierarchy is implied.

The functions `incr` and `decr` can serve as fractional access capabilities [Boyland 2003] to the counter: it is possible to pass just one of them to a client, and keep the other one hidden. A variation of this pattern can implement generic eta-expansion of mutable refs, *i.e.*, return one closure for reading and one for writing, with the same respective signatures as the primitive accessors.

2.4.2 Non-Interference. Computations involving shared state may lead to data races when run in parallel. By contrast, if two functions are separate, they cannot share any mutable state, and thus, can safely be called in parallel. Function arguments are separate by default, so we can implement a safe parallel execution operator as a higher-order function with the following signature:

```
def par(a: (() => Unit)∅)(b: (() => Unit)∅): Unit
```

The internal implementation might launch tasks for each of the two functions or otherwise orchestrate their execution, potentially using unsafe primitives. We can use `par` as follows and rely on the type system's separation guarantee to ensure non-interference:

```
val c1 = new Ref(0), c2 = new Ref(0)
par {
  c1 += ...   // ok: operate on c1 only, cannot access c2
} {
  c2 -= ...   // ok: operate on c2 only, cannot access c1
} // no interference
```

While useful, this simple pattern also has clear limits. In particular, we might want to allow read access to shared variables but disallow shared writes. This is a case where an effect system on top of reachability types is useful. We discuss such finer-grained patterns in Section 5.3.

2.4.3 Non-Escaping. Patterns that provide scoped access to a certain capability value are ubiquitous. Examples range from automatic resource management (ARM) for file or socket handles to database transactions or constructs such as exception handling. Using reachability types, we can guarantee that the introduced capability value cannot escape its scope. As an example, consider the following signature for a `try` block, implemented as a polymorphic higher-order function that introduces an exception throwing capability:

```
type CanThrow <: Exception => Nothing
def try[A∅](block: (CanThrow∅ => A∅)∅): Option[A]∅
```

The implementation can again use a variety of mechanisms, including unsafe ones, to implement the desired functionality. Given the typing of `block`, the `CanThrow` argument is tracked but cannot be aliased with the result. Hence, users are guaranteed that `throw` cannot escape the given scope:

```
val c1 = new Ref(0)
try { throw =>
  c1 += ...                          // ok: block is tracked; it can use other tracked vars
  if (error)
    throw(new Exception("failed"))
  () => throw(new Exception("boo!")) // error: block result is not allowed to alias throw
}
```

In Section 5.1, we consider a generalization of this pattern to algebraic effects and handlers.

2.4.4 Non-Accessibility. Just as it is useful to guarantee that certain values cannot escape outside a given scope, it is often useful to render certain values inaccessible within a given scope. In the simplest case, we can disable all tracked values by demanding the scope to be an untracked function,

but we can also exclude certain values specifically. Consider, *e.g.*, a global `CanIO` capability, and a higher-order function to disable it for a given scope:

```
val canIO: CanIO
def withoutIO[A^∅](cap: CanIO^∅)(block: (() => A^∅)^∅): A^∅
```

Assuming that there is only one `CanIO` value in scope, requiring it as an additional argument that must be separate from `block` renders it inaccessible within the scope represented by `block`:

```
val c1 = new Ref(0)
withoutIO(canIO) {
  c1 += ...               // ok: still allowed to use tracked vars
  readFile(path)(canIO)   // error: canIO not accessible here
}
```

The pattern can also be adapted to work with scoped instead of global capabilities, *e.g.*, to properly thread `CanThrow` capabilities through nested `try` blocks.

This pattern is handy to temporarily disable access to certain values, but sometimes we want to disable access permanently, such as for capabilities that may only be used once. We could achieve a variant of that by transforming the program to continuation-passing or monadic style, expanding the dynamic scope all the way to the end of the program, but a more direct solution is based on flow-sensitive "kill" effects (see Section 4.3).

2.4.5 Borrowing. As a variant of making values inaccessible, we sometimes want to make a value accessible in a more restricted way only. This leads to a general pattern for "borrowing" a specific value, exposing it via a new, uniquely accessible reference, within a given scope. The general interface is the following:

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

Since `x` and `block` are separate, the term `block(x)` typechecks. We can now use `borrow` as follows:

```
val c1 = new Ref(0)
borrow(c1) { c2 =>
  ... // c1 is not accessible directly here, only via c2
}
```

Many interesting variations of this pattern are possible. For example, we can implement immutable borrowing by exposing not the mutable reference itself, but a wrapper that only permits reads, following the fractional capabilities pattern from Section 2.4.1.

Ownership transfer, move semantics, etc., are essentially flow-sensitive variations of this pattern, which we support using flow-sensitive effects on top of the basic reachability system (see Section 4.3).

2.4.6 Key Take-Aways. Before we dive into the formal model, we summarize the key design points of λ^* as follows: (1) The reachable set of a function is the combined reachable set of its free variables. (2) Function arguments are checked to be separate from any of the function's free variables, unless declared otherwise. (3) Dependent applications enable function results to track the argument at the call site. (4) Self-references in function types enable tracking escaping functions that continue to interact with their captured environments; references to free variables in the type of an escaping closure are rewired to self-references to the closure itself.

3 FORMAL MODEL

This section presents a formalization of our reachability type system λ^*, a simply-typed λ-calculus with mutable reference cells and reachability qualifiers.

3.1 Syntax

Fig. 2 defines the term and type syntax of λ^*. A term t is either a constant c belonging to a base type B (*e.g.*, integers, the unit value, etc.), a variable x, a recursive lambda abstraction $\lambda f(x).t$ binding

$$
\begin{array}{lll}
t & ::= c \mid x \mid \lambda f(x).t \mid t_1\ t_2 \mid \mathsf{ref}\ t \mid !t \mid t_1 := t_2 & x, y, \ldots, f, g, \ldots \in \mathsf{Var} \\
T & ::= B \mid \mathsf{Ref}\ T \mid f(x : T^q) \to T^q & \alpha, \beta, \gamma \in \mathcal{P}_{\mathsf{fin}}(\mathsf{Var}) \\
\Gamma & ::= \varnothing \mid \Gamma, x : T^q & q ::= \bot \mid \alpha
\end{array}
$$

Fig. 2. The syntax of λ^*.

$\boxed{\Gamma \vdash t : T^q}$

$$
\textsc{T-cst}\ \frac{c \in B}{\Gamma \vdash c : B^\bot} \qquad
\textsc{T-var}\ \frac{\Gamma(x) = T^q}{\Gamma \vdash x : T^q} \qquad
\textsc{T-ref}\ \frac{\Gamma \vdash t : T^\bot}{\Gamma \vdash \mathsf{ref}\ t : (\mathsf{Ref}\ T)^\varnothing}
$$

$$
\textsc{T-assign}\ \frac{\Gamma \vdash t_1 : (\mathsf{Ref}\ T)^q \quad \Gamma \vdash t_2 : T^\bot}{\Gamma \vdash t_1 := t_2 : \mathsf{Unit}^\bot} \qquad
\textsc{T-deref}\ \frac{\Gamma \vdash t : (\mathsf{Ref}\ T)^q}{\Gamma \vdash !t : T^\bot}
$$

$$
\textsc{T-abs}\ \frac{F = f(x : T_1^{q_1}) \to T_2^{q_2} \quad (\Gamma, f : F^{q_f + f}, x : T_1^{q_1 + x})^{q_f \sqcup \{f,x\}} \vdash t : T_2^{q_2}}{\Gamma \vdash \lambda f(x).t : F^{q_f}}
$$

$$
\textsc{T-app}\ \frac{\Gamma \vdash t_1 : (f(x : T_1^{q_1 \sqcap q_f}) \to T_2^{q_2})^{q_f} \quad \Gamma \vdash t_2 : T_1^{q_1} \qquad x, f \notin FV(T_2)}{\Gamma \vdash t_1\ t_2 : T_2^{q_2[q_1/x,\ q_f/f]}}
$$

$$
\textsc{T-sub}\ \frac{\Gamma \vdash t : T_1^{q_1} \quad \Gamma \vdash T_1^{q_1} <: T_2^{q_2}}{\Gamma \vdash t : T_2^{q_2}}
$$

$\boxed{\Gamma \vdash T_1^{q_1} <: T_2^{q_2}}$

$$
\textsc{S-ref}\ \frac{\Gamma \vdash q_1 <: q_2 \qquad \Gamma \vdash T_1^\bot <: T_2^\bot \qquad \Gamma \vdash T_2^\bot <: T_1^\bot}{\Gamma \vdash (\mathsf{Ref}\ T_1)^{q_1} <: (\mathsf{Ref}\ T_2)^{q_2}} \qquad
\textsc{S-base}\ \frac{\Gamma \vdash q_1 <: q_2}{\Gamma \vdash B^{q_1} <: B^{q_2}}
$$

$$
\textsc{S-fun}\ \frac{\Gamma \vdash q_5 <: q_6 \qquad \Gamma \vdash T_3^{q_3} <: T_1^{q_1} \qquad \Gamma, f : (f(x : T_1^{q_1}) \to T_2^{q_2})^{q_5 + f}, x : T_3^{q_3 + x} \vdash T_2^{q_2} <: T_4^{q_4}}{\Gamma \vdash (f(x : T_1^{q_1}) \to T_2^{q_2})^{q_5} <: (f(x : T_3^{q_3}) \to T_4^{q_4})^{q_6}}
$$

Fig. 3. Typing and subtyping rules of λ^*.

$$
\begin{array}{lll}
C & ::= \square \mid C\ t \mid v\ C \mid \mathsf{ref}\ C \mid !C \mid C := t \mid v := C & l \in \mathsf{Loc} \\
v & ::= \lambda f(x).t \mid c \mid l \mid \mathsf{unit} & \sigma ::= \varnothing \mid \sigma, l \mapsto v \\
t & ::= \ldots \mid l &
\end{array}
$$

$\boxed{t \mid \sigma \to t' \mid \sigma'}$

$$
\begin{array}{llll}
C[(\lambda f(x).t)\ v] \mid \sigma & \to & C[t[v/x, (\lambda f(x).t)/f]] \mid \sigma & [\beta] \\
C[\mathsf{ref}\ v] \mid \sigma & \to & C[l] \mid (\sigma, l \mapsto v) \quad l \notin dom(\sigma) & [\textsc{Ref}] \\
C[!l] \mid \sigma & \to & C[\sigma(l)] \mid \sigma \quad l \in dom(\sigma) & [\textsc{Deref}] \\
C[l := v] \mid \sigma & \to & C[\mathsf{unit}] \mid \sigma[l \mapsto v] \quad l \in dom(\sigma) & [\textsc{Assign}]
\end{array}
$$

Fig. 4. Reduction Semantics of λ^*.

the function self-reference f and parameter x in body t, application $t_1\ t_2$, a reference cell ref t initialized with t, dereference $!t$, or assignment $t_1 := t_2$. We stipulate Barendregt [1985]'s convention on variables, *i.e.*, variable bindings are distinct from each other, and renamed on the fly, so that substitution is capture-free. We use $f, g, h, \ldots$ to specifically refer to function self-references.

We annotate types and typing assumptions with reachability qualifiers q. A qualifier either indicates that a term is untracked ($\bot$), or that it may alias a finite set of free variables α. Qualifiers are always well-scoped, *i.e.*, they may contain only variables bound in the context. This extends to qualifiers occurring in types, inducing a notion of term and type well-formedness.

The types consist of base types B (Int, Unit, ...), reference types Ref T, and function types $f(x : T_1^{q_1}) \to T_2^{q_2}$. Functions may be dependent, in the sense that the argument x and self-reference f may occur in the codomain's qualifier q_2 (see Section 2.3). Plus, x may also occur in any qualifier within T_2 itself. For example, $f(x : (\mathsf{Ref}\ \mathsf{Int})^\varnothing) \to (\mathsf{Ref}\ \mathsf{Int})^{\{x,f\}}$ is a legal function type. Self-references behave in the same way as self-types in Scala and DOT [Amin et al. 2016], although they are treated as part of function types in our formalization. This helps to reduce the number of

rules in the calculus, but we could equally well include the regular DOT self-types $\mu x.T$ as separate entities in the type language. Either way, the function type above can be read as syntactic sugar for $\mu f.(x : (\text{Ref Int})^{\varnothing}) \to (\text{Ref Int})^{\{x,f\}}$.

3.2 Reachability Qualifiers and Operations

The inclusion order on qualifiers $q_1 \sqsubseteq q_2 := q_1 = \bot \vee q_1 \subseteq q_2$ is the partial order that includes the subset relation on variable sets, enriched with the untracked qualifier $\bot$ as the least element. This relation induces a union operation $q_1 \sqcup q_2$ on qualifiers. We write $q + x$ for decomposing/pattern matching qualifiers and union of qualifiers with variables. In the latter case, $+$ acts as a union where the untracked qualifier $\bot$ is cancelling: $\bot + x = \bot$ and $\alpha + x = \alpha \sqcup \{x\}$. This avoids duplicating rules for tracked and untracked cases. In addition, we write $q_1 \oplus q_2$ for the general cancelling union on qualifiers: $\bot \oplus q = \bot$ and $\alpha \oplus q = \alpha \sqcup q$. It is useful to define the intersection $\sqcap$ so that the least element $\bot$ removes all elements from sets, *i.e.*, $\alpha \sqcap \bot = \bot \sqcap \alpha = \varnothing$, $\bot \sqcap \bot = \bot$, and $\alpha \sqcap \beta = \alpha \cap \beta$.

The outer qualifier $f(\ldots)^q$ of a function type indicates that its values may close over all the variables in the context whose reachability qualifier is at most q, expressed by an environment filter: $\Gamma^q := \{x : T^{q'+x} \in \Gamma \mid q' + x \sqsubseteq q\}$. Consequently, when typing abstractions $\lambda f(x).t$, the smallest q we can assign is the union of the free variables $FV(t) \setminus \{f, x\}$ and their assumed qualifiers. As an invariant, we stipulate that typing assumptions in Γ are always of the form $x : T^{q+x}$, *i.e.*, if the assumption is tracked ($q \neq \bot$), then the variable x is included in the qualifier.

3.3 Type Assignment

The typing judgment $\Gamma \vdash t : T^q$ in Fig. 3 states that the term t has type T and may reach (all variables in) q under context Γ. Ignoring qualifiers, we have a standard type system for the simply-typed λ-calculus with recursive functions, mutable references, and subtyping. We also implicitly assume well-scopedness of types and qualifiers for all judgment forms under the typing context Γ.

Constants of a base type are untracked values (T-CST). We restrict reference introduction (T-REF) to untracked values only, and qualify the reference itself as tracked/fresh, effectively prohibiting nested references. Accordingly, the right-hand side of an assignment (T-ASSIGN) must be untracked, and dereferencing yields untracked values (T-DEREF). We relax this restriction to allow nested references in Section 4. An alternative way of expressing the type of a fresh reference would be using self-types, *i.e.*, $\Gamma \vdash \text{ref } t : \mu z.(\text{Ref } T)^{\{z\}}$ instead of $(\text{Ref } T)^{\varnothing}$. The latter is more lightweight, but can be regarded as syntactic sugar for the former (see Section 4.4).

The rules for abstraction (T-ABS) and application (T-APP) govern observable separation and overlap among functions, their arguments, and the environment (see Section 2.2). The reachability qualifier q_f in the function type is an upper bound on the combined reachable set of the function's free variables, describing what the function implementation can and cannot access from the environment. This is enforced by imposing an environment filter (see Section 3.2) in the typing of the function body (T-ABS). We filter the environment with $q_f \sqcup \{x, f\}$ which includes the self-reference f (since functions are recursive), and the argument variable x. The qualifier q_1 represents what the function can observe at most about its argument x, and it is only accessible if q_f includes q_1. Any other reachability information provided by callers of the function is not visible to its implementation and is thus separate. In function application (T-APP), this principle manifests in form of the intersection constraint $q_1 \sqcap q_f$ on the function domain, relating the function qualifier q_f (definition site) and the provided argument's qualifier q_1 (call site). It means that the function's computation is not affected by anything out of $q_1 \setminus q_f$ which can be viewed as a form of implicit contextual polymorphism (see Sections 3.4 and 5.1). Furthermore, we support dependent function application to merge the reachability information from the definition site with the information from the call site. If the function result $T_2^{q_2}$ is dependent on the argument x or the self-reference f, then these are substituted

with the respective annotations of the function and argument. Dependency on x is shallow, in the sense that it should not occur freely in a nested qualifier within T_2. Application is restricted in this way for the reasons we motivated in Section 2.3, *i.e.*, to correctly model escaping closures, we abstract over nested occurrences of x within T_2 using self-references and subtyping.

Finally, we may assign a less precise type and qualifier to a term by subsumption (T-SUB).

3.4 Lightweight Reachability Polymorphism

Dependent application (T-APP) substitutes the function's formal parameter with the argument's qualifier, yielding a form of lightweight qualifier polymorphism:

```
// assume c1 : Ref[Int]^{a,b,c1} and c2 : Ref[Int]^{c2}
def inc(x: Ref[Int]^∅) = { x := !x + 1; x } // : ((x : Ref[Int]^∅) => Ref[Int]^{x})^⊥
(inc(c1), inc(c2), inc(new Ref(0)))        // : (Ref[Int]^{a,b,c1}, Ref[Int]^{c2}, Ref[Int]^∅)
```

However, this substitution-based polymorphism cannot abstract over tracked vs. untracked variables, *e.g.*, consider the following function returning a fresh reference:

```
// : (T^∅ => Ref[Int]^∅)^⊥ <: ((x : T^∅) => Ref[Int]^{x})^⊥
def f(x: T^∅) = { new Ref(0) }
```

Function `f` can be upcast to a dependent function via subtyping (see Section 3.5). If we call `f` with an untracked $\bot$ argument, we might expect the result type `Ref[Int]`$^{\{x\}[\bot/x]}$ `= Ref[Int]`$^{\bot}$. While this would be desirable from the point of view of polymorphism, it would incorrectly mark the returned reference as untracked. Hence, we just remove variables from sets when they are substituted with $\bot$, *i.e.*, `Ref[Int]`$^{\{x\}[\bot/x]}$ `= Ref[Int]`$^{\varnothing}$ which makes qualifier polymorphism non-parametric. We discuss further extensions with parametric polymorphism in Section 3.9.

3.5 Subtyping

Modulo qualifiers, subtyping for types $\Gamma \vdash T_1^{q_1} <: T_2^{q_2}$ is standard, *i.e.*, function types are contravariant in their domain, and covariant in the codomain (S-FUN), which extends analogously to qualifiers. Reference types are covariant in their qualifiers, and invariant in their type argument (S-REF). While subtyping allows untracked reference types, the typing rules prohibit introducing these. Such reference types may only appear in absurd assumptions, indicating dead code. Untracked base types can be treated as tracked (S-BASE). Subtyping on qualifiers $\Gamma \vdash q_1 <: q_2$ generalizes the order $\sqsubseteq$ (Section 3.2) to include abstraction by function self-references and enforces well-scopedness under Γ. It permits function self-references to serve as an abstraction on a closure's captured variables. Recall the example of escaping closures in Section 2.3, where we rewire references to free variables in the type of an escaping closure to self-references to the closure itself. For a function type $(f(\ldots) \to \ldots)^q$ in Γ, we treat f and $q + f$ as equivalent, *i.e.*, $\Gamma \vdash \{f\} <: q + f$ and $\Gamma \vdash q + f <: \{f\}$.

3.6 Reduction Semantics

We formalize the dynamic semantics of λ^* in terms of a single-step reduction relation (Fig. 4), which is entirely standard [Pierce 2002]. For the semantics of references, we enrich the term language with store locations l and classify these as values along with functions and constants, including the unit value. Redexes consist of a term and a store σ, which is a finite map from locations to values. The typing relation now requires an additional store typing Σ, mapping locations to base types with $\bot$ qualifiers. Plus, reachability qualifiers may now also include location values. All of these extensions are benign and we elide them to not clutter the presentation. One thing of note is that locations l track themselves, *i.e.*, if $\Sigma(l) = T^{\bot}$, then $\Gamma \mid \Sigma \vdash l : (\mathrm{Ref}\ T)^{\{l\}}$. To connect static and dynamic semantics, we require the usual relation $\Gamma \mid \Sigma \vdash \sigma$ stating that $dom(\Sigma) = dom(\sigma)$ and $\Gamma \mid \Sigma \vdash \sigma(l) : \Sigma(l)$ for each $l \in dom(\sigma)$.

3.7 Type Soundness

We prove type safety of λ^* by the standard syntactic soundness approach [Pierce 2002; Wright and Felleisen 1994]. Type safety statically asserts that well-typed programs never get stuck and also ensures *reachability-safety*: if two well-typed programs have disjoint reachability qualifiers, then their final answers will not share any aliasing. Since λ^* tracks variables, it has a form of dependent types, and we require a substitution lemma that also takes qualifiers into account. Here, we state the lemma in a form that mirrors the conditions of the type preservation proof in the case of call-by-value β reduction (Figure 4), substituting an argument value into a function body along with the function itself. The proof requires a suitably generalized induction hypothesis:

Lemma 3.1 (Substitution). *If* $\varnothing \mid \Sigma \vdash v : T_1^{q_1}$, *and* $\varnothing \mid \Sigma \vdash \lambda f(x).t_1 : (f(x{:}T_1^{q_1 \sqcap q_f}) \to T_2^{q_2})^{q_f}$, *and* $x, f \notin FV(T_2)$, *then* $\varnothing \mid \Sigma \vdash t_1[v/x, (\lambda f(x).t_1)/f] : T_2^{q_2[q_1/x, q_f/f]}$.

Importantly, the lemma accounts for the separation between argument and function enforced by the typing rule T-App (see Figure 3), *i.e.*, the function cannot observe anything in the argument's qualifier q_1 beyond the intersection $q_1 \sqcap q_f$. To see why, observe that we substitute the function f into its own body t_1, and by the given typing evidence for f, we know that t_1 cannot access anything in the typing context Γ that is not included by the qualifier q_f (see T-Abs in Figure 3). Therefore, the body only observes the common overlap between q_1 and q_f.

Theorem 3.2 (Progress). *If* $\varnothing \mid \Sigma \vdash t : T^q$, *then either* t *is a value, or for any store* σ *such that* $\varnothing \mid \Sigma \vdash \sigma$, *there exists a term* t' *and store* σ' *such that* $t \mid \sigma \to t' \mid \sigma'$.

The preservation theorem guarantees that the underlying type T is preserved, and the store may increase in size during reduction.

Theorem 3.3 (Preservation). *If* $\varnothing \mid \Sigma \vdash t : T^q$, *and* $\varnothing \mid \Sigma \vdash \sigma$, *and* $t \mid \sigma \to t' \mid \sigma'$, *then there exists* $\Sigma' \supseteq \Sigma$ *and* $q' \sqsubseteq dom(\Sigma') \setminus dom(\Sigma)$ *such that* $\varnothing \mid \Sigma' \vdash \sigma'$ *and* $\varnothing \mid \Sigma' \vdash t' : T^{q \oplus q'}$.

The proof of the preservation theorem establishes that allocation (ref t) is the only construct that extends the store. Thus, two freshly allocated locations induced by different reduction steps must be disjoint. This observation gives us the following corollary that makes the notion of reachability-safety precise, guaranteeing that if two sets of reachable variables are disjoint ($q_1 \sqcap q_2 \sqsubseteq \varnothing$) before reduction, then the disjointness is preserved afterwards ($q_1' \sqcap q_2' \sqsubseteq \varnothing$):

Corollary 3.4 (Preservation of Separation). *If* $\varnothing \mid \Sigma \vdash t_1 : T_1^{q_1}$, *and* $\varnothing \mid \Sigma \vdash t_2 : T_2^{q_2}$, *and* $q_1 \sqcap q_2 \sqsubseteq \varnothing$, *and* $\varnothing \mid \Sigma \vdash \sigma$, *and* $t_1 \mid \sigma \to t_1' \mid \sigma'$, *and* $t_2 \mid \sigma' \to t_2' \mid \sigma''$, *then there exists* $\Sigma' \supseteq \Sigma$, *and* $\Sigma'' \supseteq \Sigma'$, *and* q_1', q_2', *such that* $\varnothing \mid \Sigma' \vdash t_1' : T_1^{q_1'}$, *and* $\varnothing \mid \Sigma'' \vdash t_2' : T_2^{q_2'}$, *and* $q_1' \sqcap q_2' \sqsubseteq \varnothing$.

3.8 Encodings

Although the core language is compact, it is able to encode a number of common language constructs as derived syntax, some of which were already used in Section 2.

Let-Bindings. Abstractions and function applications can encode Let-bindings in the standard way, *i.e.*, $\mathsf{let}\ x = t_2\ \mathsf{in}\ t_1 \triangleq (\lambda f(x).t_1)\ t_2$. The typing rule for let is shown below:

$$\frac{\Gamma, x : T_1^{q_1 + x} \vdash t_1 : T_2^{q_2} \qquad \Gamma \vdash t_2 : T_1^{q_1} \qquad x \notin FV(T_2)}{\Gamma \vdash \mathsf{let}\ x = t_2\ \mathsf{in}\ t_1 : T_2^{q_2[q_1/x]}}\ \textsc{Let-Encoding}$$

When translating lets to applications, we assume a fresh function name f for the lambda, which does not appear in the let's body. This ensures the result type of the translated application $T_2^{q_2[q_1/f, q_1/x]}$ is equivalent to the type $T_2^{q_2[q_1/x]}$ of the let expression.

Multi-Argument Functions. We treat multi-argument functions of the form

$$(f(x_1 : A_1^{q_1}, \ldots, x_n : A_n^{q_n}) \to B^{q_{n+1}})^{q_f}$$

and their application as syntactic sugar for their curried versions

$$(f(x_1 : A_1^{q_1}) \to (g_2(x_2 : A_2^{q_2}) \to \ldots (g_n(x_n : A_n^{q_n}) \to B^{q_{n+1}})^{q_{g_n}} \ldots)^{q_{g_2}})^{q_f}$$

such that $g_2, \ldots, g_n$ are fresh, and the inner function qualifiers $q_{g_2}, \ldots, q_{g_n}$ contain the multi-argument function's qualifier q_f. As a special case, zero-argument functions desugar to functions with a $\mathsf{Unit}^{\perp}$ argument.

Algebraic Data Types. Algebraic data types can be encoded à la Church or Scott, with the addition of qualifiers in types. For instance, Church pairs and sums in λ^* have these type shapes:

$$\mathsf{Pair}^{q_1 \cdots q_7}[A, B]_X := ((A^{q_1} \to (B^{q_2} \to X^{q_3})^{q_4})^{q_5} \to X^{q_6})^{q_7}$$
$$\mathsf{Sum}^{q_1 \cdots q_9}[A, B]_X := ((A^{q_1} \to X^{q_2})^{q_3} \to ((B^{q_4} \to X^{q_5})^{q_6} \to X^{q_7})^{q_8})^{q_9}$$

Since the base system lacks polymorphism, these types are families of distinct monomorphic instances, parameterized at the meta level over the elimination type X and qualifiers (cf. Section 3.9). To illustrate the need for so many qualifiers, consider the first and second projections on pairs:

$$\pi_1^{q_1,q_2}_{A,B} \quad := \quad \lambda f(a).\lambda g(b).a \quad : \quad (f(a : A^{q_1}) \to (g(b : B^{q_2}) \to A^{\{g\}})^{q_1+a})^{q_1}$$
$$\pi_2^{q_1,q_2}_{A,B} \quad := \quad \lambda f(a).\lambda g(b).b \quad : \quad (f(a : A^{q_1}) \to (g(b : B^{q_2}) \to B^{q_2+b})^{q_2})^{\emptyset}$$

The qualifiers in the dependent function types reveal which components are projected and are thus non-uniform. Notably, the first projection π_1 returns a value of type $A^{\{g\}}$, because the inner function closes over the first component, and λ^* does not support full dependent application.

3.9 Extensions

Track/Non-Track Distinction, Mutable References. Although we chose to present the type system including non-tracked values and mutable references, these can be viewed as extensions. The type system could be useful without them, *e.g.*, for tracking capabilities or for memory management in functional high-performance DSLs [Rompf et al. 2015].

Parametric Polymorphism. It is natural to extend the base type system with universal quantification over types as in System F [Girard 1971; Reynolds 1974], *e.g.*, $\forall X.(X^q \to X^q)^q$, where X is a type variable. This is unproblematic, as long as the $\forall$ only quantifies over types, but not qualifiers. Unbounded quantification over qualifiers would imply arbitrary potential aliasing, largely defeating the purpose of the system. Hence, an extension with parametric qualifier polymorphism should use bounded quantification in the style of System $F_{<:}$ [Cardelli et al. 1991], *e.g.*, $\forall \rho <: \alpha.T^{\rho}$. This is again straightforward as long as the qualifier variable ρ is implicitly lower-bounded by $\emptyset$, *i.e.*, may only range over sets of variables, but not $\perp$, for the same reason that dependent application in the λ^* base system is restricted in a similar way (see Section 3.4). Hence, the expressiveness gains of this form of qualifier polymorphism are not entirely clear. Abstracting over tracked and non-tracked status via qualifier variables with lower bound $\perp$ requires more involved treatment.

Must-Reach Tracking. The reachability qualifiers are an upper bound on the set of reachable values. We can extend the system with lower bounds, *i.e.*, $T^{l..u}$, where l and u are the sets of *must-reachable* variables and *may-reachable* variables, respectively. As noted above, distinguishing lower bounds $\perp$ and $\emptyset$ would also enable advanced forms of polymorphism. Must-reachable qualifiers behave in the opposite way as may-reachable qualifiers, *e.g.*, when joining alternative control branches, intersection and union are used for computing the lower and upper bound, respectively. With a flow-sensitive effect system (Section 4.4), the must-reach information can enable tracking effects such as object initialization [Kabir et al. 2020; Liu et al. 2020; Summers and Müller 2011].

Strict vs Non-Strict Reachability. The λ^* system can only express that a variable may point to an object, but it cannot distinguish if the variable points to the object itself or if the object is reachable indirectly through object fields or other means of dereference. We can gain additional precision by distinguishing between strict and non-strict reachability, recording the number of dereferences in qualifiers (*e.g.*, 0, 1, 0 or more, 1 or more) [Odersky 1991]. Together with must-reachability qualifiers, we can encode Scala's singleton types [Odersky and Rompf 2014]. Further extensions could include field names for dereferenced records.

Inverse Reachability. Reachability qualifiers represent unidirectional paths from a variable to its reachable values. We can add qualifiers for inverse reachability, *i.e.*, tracking incoming as opposed to outgoing references and following reachability paths in the reverse direction of pointers. This allows us to track that an object is contained within another. Together with the strict reachability qualifiers, we can ensure termination of Scott encodings, *i.e.*, recursion on strict subvalues.

4 THE MARRIAGE OF EFFECTS AND REACHABILITY

The reachability types of λ^* (Section 3) are already quite expressive, reconciling higher-order with stateful programming. Yet, we have not "reached" the end of the line, and there is ample grounds to make the type system reflect more nuanced usage patterns. For example, the ubiquitous read and write capability pattern shown in Section 2 suggests that it is useful to discern reads and writes on variables at the type level. More generally, this section shows how a generic effect system on variable sets neatly complements reachability types.

To ensure safety, the λ^* type system prohibits nested references, permitting only untracked values in reference cells. We may recover general nested references from a *flow-sensitive* effect system with move semantics. For example, the following program

```
def f(x: Ref[Int]^∅) = { val y = move(x); ... }
val z = new Ref(1); f(z) // z is "killed" by f and cannot be used anymore
```

defines a function `f`, whose body *uniquely* owns the referent of `x` through its local variable `y`. The ownership transfers from `x` to `y` via the operator `move`, which kills other aliases of `x` (including `x` itself). Then `f` is applied to a resource `z` allocated later. The ownership transfer is flow-sensitive, in the sense that the move operation affects subsequent execution steps. Since `z` and `x` are aliased, `z` is killed too and cannot be used after the application `f(z)`.

With flow-sensitive effects and reachability types, we can readily express use-once capabilities in a direct style. For example, a function of type/effect $f(x : T_1^{q_1}) \xrightarrow{\{(f,\mathsf{kill})\}} T_2^{q_2}$ induces a latent effect that "kills" itself, effectively disallowing future invocations of the function after its first use. The annotation $\{(f, \mathsf{kill})\}$ above the arrow denotes the latent effect induced by calling this function, which makes use of our notion of function self-references f (Section 3).

The ownership transfer induces a "kill" effect, but we have not specified what should be disallowed after a variable is killed. There are multiple options. Consider the following program that transfers the ownership of the reference from `x` to `y` via the `move` operator, which explicitly "kills" variable `x`.

```
val x = new Ref(1); val y = move(x) // transfer ownership of x to y
x += 1 /* use x */; f(x) /* mention x */
```

It is clear that the update operation `x += 1` should be prohibited, which *uses* `x` in a relevant and critical way. However, passing the dangling pointer to a function `f(x)` (*i.e.*, *mentioning* `x`) is arguably benign as long as the subsequent evaluation of `f` does not *use* `x` (see Section 4.4 for further discussion). By discerning this subtlety [Gordon 2020a], we are able to classify effects into "gen", "use", or "kill" categories (akin to dataflow analysis).

The rest of this section discusses the effect system λ^*_ϵ as an extension of λ^*, which is built on top of the *effect quantales* algebraic structure by Gordon [2021]. Section 4.1 reviews the generic

effect quantales framework. We contribute a novel *store-sensitive effect quantale* structure based on reachability types. Then we demonstrate two instantiations: (1) Section 4.2 extends the type system from Section 3 to a *flow-insensitive* effect system that models "use" (read and write) effects. (2) Section 4.3 extends the language with move and swap constructs, and presents a flow-sensitive instantiation with higher-order state and ownership transfer semantics. To properly model the ownership transfers semantics, we use a destructive "kill" effect in Section 4.3. Finally, we discuss possible extensions to model "gen" effects (such as object initialization) in Section 4.4.

4.1 Generic Framework

We start from the definition of effect quantale, which serves as the foundation of later formulations.

Definition 4.1 (Effect Quantale [Gordon 2021]). An effect quantale $(E, \sqcup, \triangleright, I)$ is a partial (binary) join semilattice $(E, \sqcup)$ with partial monoid $(E, \triangleright, I)$.

The *join operator* $\sqcup : E \times E \to E$ from the semilattice combines effects from alternative control branches, for example, at the join point of a conditional statement. The operator $\sqcup$ is commutative, *i.e.*, $e_1 \sqcup e_2 = e_2 \sqcup e_1$. We use the *sequential composition operator* $\triangleright : E \times E \to E$ from the monoid to compose effects when the order of evaluation matters. It is generally not commutative. Both $\sqcup$ and $\triangleright$ are partial functions – they do not have to be defined for all elements in E.

The interaction between join and sequential composition requires an additional premise: $\triangleright$ must distribute over $\sqcup$ in both directions, *i.e.*, $a \triangleright (b \sqcup c) = (a \triangleright b) \sqcup (a \triangleright c)$ and $(a \sqcup b) \triangleright c = (a \triangleright c) \sqcup (b \triangleright c)$. Both sides have to be defined or undefined at the same time.

4.1.1 Store-Sensitive Effect Quantale. The λ^* type system already provides reachable information over variables (and store locations). We now would like to determine the effects caused by aliased variables. This suggests a set of mappings from aliased variables to their effects, informing a novel *store-sensitive effect quantale* that integrates reachability information. We parameterize the store-sensitive effect quantale over the choice of *effect labels* $\mathbb{E}$, which is another effect quantale and represents the possible effects modeled in the system. We instantiate $\mathbb{E}$ to flow-insensitive effects in Section 4.2 and a flow-sensitive variant in Section 4.3.

Definition 4.2. The *store-sensitive effect quantale* $(\Delta(\mathbb{E}), \sqcup, \triangleright, I)$ is defined as follows:

- $\Delta(\mathbb{E}) = \{\epsilon \in \mathcal{P}(\mathcal{P}(\mathsf{Loc} \cup \mathsf{Var}) \times \mathbb{E}) \mid \forall(\alpha_i, e_i) \in \epsilon, \forall(\alpha_j, e_j) \in (\epsilon \setminus (\alpha_i, e_i)), \alpha_i \cap \alpha_j = \emptyset\}$
  An instance $\epsilon \in \Delta(\mathbb{E})$ is a set of pairs. A pair $(\alpha, e) \in \epsilon$ denotes that the set of aliased variables α (domain) may have effect e (codomain). The domains in ϵ are always pairwise disjoint.
- The commutative join operator $\sqcup$ is defined as:
  $$\begin{array}{lll} \epsilon_1 \sqcup \emptyset & = \epsilon_1 & \\ \{(\alpha_1, e_1), \epsilon_1\} \sqcup \{(\alpha_2, e_2), \epsilon_2\} & = \{(\alpha_1 \cup \alpha_2, e_1 \sqcup_{\mathbb{E}} e_2), \epsilon_1\} \sqcup \epsilon_2 & \text{if } \alpha_1 \cap \alpha_2 \neq \emptyset \\ \epsilon_1 \sqcup ((\alpha_2, e_2), \epsilon_2) & = ((\alpha_2, e_2), \epsilon_1) \sqcup \epsilon_2 & \text{otherwise} \end{array}$$
- The sequential composition operator $\triangleright$ is defined as:
  $$\begin{array}{lll} \epsilon_1 \triangleright \emptyset & = \epsilon_1 & \\ \{(\alpha_1, e_1), \epsilon_1\} \triangleright \{(\alpha_2, e_2), \epsilon_2\} & = \{(\alpha_1 \cup \alpha_2, e_1 \triangleright_{\mathbb{E}} e_2), \epsilon_1\} \triangleright \epsilon_2 & \text{if } \alpha_1 \cap \alpha_2 \neq \emptyset \\ \epsilon_1 \triangleright ((\alpha_2, e_2), \epsilon_2) & = ((\alpha_2, e_2), \epsilon_1) \triangleright \epsilon_2 & \text{otherwise} \end{array}$$
- The ordering $\sqsubseteq$ over effects in $\Delta(\mathbb{E})$ requires information from the typing context, because effects group aliased variables into sets. First, we define the component-wise ordering over pairs $\mathcal{P}(\mathsf{Loc} \cup \mathsf{Var}) \times \mathbb{E}$, that combines qualifier subtyping and the effect label order $(\alpha_1, e_1) \leq (\alpha_2, e_2) \Leftrightarrow \alpha_1 <: \alpha_2 \wedge e_1 \sqsubseteq_{\mathbb{E}} e_2$. Then, $\epsilon_1 \sqsubseteq \epsilon_2$ is the lifting of $\leq$ to effects, *i.e.*,
  $$\epsilon_1 \sqsubseteq \epsilon_2 \Leftrightarrow \forall(\alpha_1, e_1) \in \epsilon_1, \exists(\alpha_2, e_2) \in \epsilon_2, \text{s.t. } (\alpha_1, e_1) \leq (\alpha_2, e_2).$$
- The identity element $I = \emptyset$.

The intuition behind Definition 4.2 is to track a set, whose elements are pairs of aliased variables and their corresponding effects. We maintain the additional invariant that store-sensitive effect quantales have non-overlapping domains, e.g., if $\{(\alpha, e_1), (\beta, e_2)\}$ is a store-sensitive quantale, then $\alpha \cap \beta \neq \emptyset$. Otherwise, there would be ambiguity when resolving the effect of variables. The disjointness property is preserved by the join and sequential composition operators.

When combining two effects, the join operator $\sqcup$ keeps all pairs that do not overlap in their domain, and computes the least general effect label $e_1 \sqcup_{\mathbb{E}} e_2$ for pairs that share aliased variables causing effect e_1 and e_2. Similar to the join operator, the sequential composition operator $\triangleright$ keeps all non-overlapping pairs. For overlapping pairs $(\alpha_1, e_1) \in \epsilon_1$ and $(\alpha_2, e_2) \in \epsilon_2$, where ϵ_1 is the preceding store-sensitive effect, we sequentially combine their effects $e_1 \triangleright_{\mathbb{E}} e_2$ using $\triangleright_{\mathbb{E}}$ from the parameter $\mathbb{E}$. Since $e_1 \triangleright_{\mathbb{E}} e_2$ is non-commutative and potentially undefined (to rule out invalid operations), the sequential composition of store-sensitive effects could be undefined as well. Next, we show that the effect quantale satisfies the distributivity law.

Lemma 4.3. *With respect to Definition 4.2, if $\mathbb{E}$ is an effect quantale, then* $\forall a, b, c \in \Delta(\mathbb{E}), a \triangleright (b \sqcup c) = (a \triangleright b) \sqcup (a \triangleright c)$ *and* $(a \sqcup b) \triangleright c = (a \triangleright c) \sqcup (b \triangleright c)$.

4.1.2 Type-and-Effect System. Fig. 5 shows selected type-and-effect rules for λ^*_ϵ, which extends the λ^* system (Section 3) with effects. The judgment has the form $\Gamma \mid \Sigma \vdash t : T^\alpha \mid \epsilon$, where the additional ϵ denotes the effect that may occur when evaluating t. For example, variables such as x induce a no-op effect (*i.e.*, mention) over their aliases q, as they do not induce effects on stores (shown in E-VAR). The no-op effect is represented by the least element $\bot_{\mathbb{E}}$ in the effect label. The aliases lookup operation q^* collects all the aliases that q may point to, in a transitive way. The full definition of q^* under contexts is shown in Fig. 5.

The E-ABS rule defines the type of lambda terms. Function types are slightly refactored to record the *latent* effect (*i.e.*, the ϵ above the arrow), which is the effect when calling the function. Latent effects in function types are covariant, as shown by the new subtyping rule at the top of Fig. 5. Although a function has its qualifier q_f, creating a lambda term has no store-effect over q_f, *i.e.*, it only *mentions* variables in q_f, thus we designate $\bot_{\mathbb{E}}$ in the conclusion of E-ABS.

The E-APP rule states that the effect of an application is obtained by sequentially composing multiple effects. The evaluation of sub-terms t_1 and t_2 may induce their own effects, which are called the *inherited effects* [Lucassen and Gifford 1988] of the application. The inherited effects ϵ_1 and ϵ_2 and the latent effect ϵ_3 are sequentially composed by $\triangleright$. The latent effect ϵ_3 may mention the argument name or function self-reference, too. Therefore we need to replace them with their aliased variables from the call-site, *i.e.*, $\epsilon_3[q_1^*/x, {q_f}^*/f]$. The substitution over a store-sensitive effects is defined at the bottom of Fig. 5. Subeffecting (rule E-SUB) allows a term to be considered as causing more effects than necessary with respect to the ordering relation from Definition 4.2.

Using effect quantales to formalize type-and-effect systems subsumes traditional commutative effect systems [Gordon 2021]. If we use the join operator for sequential composition, *i.e.*, $(E, \sqcup, \sqcup, I)$, then the type system is flow-insensitive, which is our first instantiation described in Section 4.2.

4.1.3 Soundness. Our type-and-effect system and the store quantale is a simply-typed instantiation of Gordon [2021]'s generic framework, which also provides a notion of abstract soundness and its proof. Intuitively, in addition to standard type safety, the soundness of the effect system implies that the static effects over-approximate the actual effects happening at runtime.

The abstract soundness is parameterized over a number of constructs, *e.g.*, the definition of quantale, runtime state, and the effects of primitive operations (such as read and write). Our semantics already provides some of these constructs, although in slightly different form. However, one missing piece is that the dynamic semantics (Section 3) is too weak to establish effect soundness,

$$\boxed{\Gamma \mid \Sigma \vdash T_1^{q_1} <: T_2^{q_2}} \quad \boxed{\Gamma \mid \Sigma \vdash t : T^{\alpha} \mid \epsilon}$$

E-FUN

$$\frac{\Gamma \mid \Sigma \vdash q_5 <: q_6 \qquad \Gamma \mid \Sigma \vdash T_3^{q_3} <: T_1^{q_1} \qquad \Gamma \mid \Sigma \vdash \epsilon_1 \sqsubseteq \epsilon_2 \qquad \Gamma, f : (f(x : T_1^{q_1}) \xrightarrow{\epsilon_1} T_2^{q_2})^{q_5+f}, x : T_3^{q_3+x} \mid \Sigma \vdash T_2^{q_2} <: T_4^{q_4}}{\Gamma \mid \Sigma \vdash (f(x : T_1^{q_1}) \xrightarrow{\epsilon_1} T_2^{q_2})^{q_5} <: (f(x : T_3^{q_3}) \xrightarrow{\epsilon_2} T_4^{q_4})^{q_6}}$$

E-VAR

$$\frac{\Gamma(x) = T^q}{\Gamma \mid \Sigma \vdash x : T^q \mid \{(q^*, \bot_{\mathbb{E}})\}}$$

E-ABS

$$\frac{F = f(x : T_1^{q_1}) \xrightarrow{\epsilon} T_2^{q_2} \qquad (\Gamma, f : F^{q_f+f}, x : T_1^{q_1+x})^{q_f \sqcup \{f,x\}} \mid \Sigma \vdash t : T_2^{q_2} \mid \epsilon}{\Gamma \mid \Sigma \vdash \lambda f(x).t : F^{q_f} \mid \{({q_f}^*, \bot_{\mathbb{E}})\}}$$

E-APP

$$\frac{\Gamma \mid \Sigma \vdash t_1 : (f(x : {T_1}^{q_1 \sqcap q_f}) \xrightarrow{\epsilon_3} {T_2}^{q_2})\ {q_f} \mid \epsilon_1 \qquad \Gamma \mid \Sigma \vdash t_2 : T_1^{q_1} \mid \epsilon_2 \qquad x, f \notin FV(T_2)}{\Gamma \mid \Sigma \vdash t_1\ t_2 : T_2^{q_2[q_1/x, q_f/f]} \mid \epsilon_1 \rhd \epsilon_2 \rhd \epsilon_3[{q_1}^*/x, {q_f}^*/f]}$$

E-SUB

$$\frac{\Gamma \mid \Sigma \vdash t : T^q \mid \epsilon_1 \qquad \Gamma \mid \Sigma \vdash \epsilon_1 \sqsubseteq \epsilon_2}{\Gamma \mid \Sigma \vdash t : T^q \mid \epsilon_2}$$

$$\begin{array}{lll}
_[_/_] & : \epsilon \to q \to \mathsf{Var} \to \epsilon & \\
\varnothing[q/x] & = \varnothing & \\
\{(\alpha, e), \epsilon\}[q/x] & = \{(\alpha[q/x], e), \epsilon[q/x]\} & \\
_ \vdash _^* & : \Gamma \to q \to q & \\
\Gamma \vdash \bot^* & = \varnothing & \\
\Gamma \vdash \alpha^* & = \Gamma \vdash \varnothing\ \alpha^* & \\
_ \vdash _\ _^* & : \Gamma \to \alpha \to \alpha \to \alpha & \\
\Gamma \vdash \alpha\ \varnothing^* & = \alpha & \\
\Gamma \vdash \alpha\ (x, \beta)^* & = \Gamma \vdash (x, \alpha)\ ((\beta \cup \gamma) \setminus (x, \alpha))^* & \\
 & \quad \text{if } x : T^{(x,\gamma)} \in \Gamma & \\
\Gamma \vdash \alpha\ (x, \beta)^* & = \Gamma \vdash (x, \alpha)\ (\beta \setminus (x, \alpha))^* & \\
 & \quad \text{otherwise} &
\end{array}$$

Fig. 5. The generic type-and-effect inference rules (excerpted) and the definitions of auxiliary functions (substitution over store-sensitive effect quantale, transitive aliases lookup).

due to the lack of tracking the runtime effects. In the following, we sketch an instrumented dynamic semantics that records runtime effects, which can be leveraged to establish soundness.

The general form of the instrumented dynamic semantics now becomes $t \mid \sigma \to^{\epsilon} t' \mid \sigma'$, where $\epsilon \in \Delta(\mathbb{E})$ is the actual effect induced by the reduction step. Then we can express the proposition of single-step type-and-effect preservation as follows:

Proposition 4.4 (Preservation, adapted from [Gordon 2021]). *If* $\Gamma \mid \Sigma \vdash t : T^q \mid \epsilon_1$, *and* $\Gamma \mid \Sigma \vdash \sigma$, *and* $t \mid \sigma \to^{\epsilon_2} t' \mid \sigma'$, *then there exists* $\Sigma' \supseteq \Sigma$, *and* $q' \sqsubseteq dom(\Sigma') \setminus dom(\Sigma)$ *such that* $\Gamma \mid \Sigma' \vdash \sigma'$, *and* $\Gamma \mid \Sigma' \vdash t' : T^{q \oplus q'} \mid \epsilon_3$, *and* $\epsilon_2 \rhd \epsilon_3 \mid_{dom(\Sigma)} \sqsubseteq \epsilon_1$.

Informally, Proposition 4.4 states that the sequential composition ($\epsilon_2 \rhd \epsilon_3 \mid_{dom(\Sigma)}$) of the effect from the single-step reduction and the static effect of the contractum (modulo new locations) are restricted by the static effect ϵ_1.

4.2 Tracking Read/Write with Flow-Insensitive Instantiation

The effects of a term provide a summary of the term's behavior when evaluated. For example, the language introduced in Section 3 exhibits read and write effects: $!t$ reads the value from the location from evaluating t and $t_1 := t_2$ writes the value of t_2 into the store. In this section, we instantiate the generic framework to a flow-insensitive system that utilizes reachability information and precisely tracks the read/write effects of terms. The effect labels of read/write effect $\mathbb{E}_{rw}$ are defined as follows:

Definition 4.5 (Effect Label). The effect label set $\mathbb{E}_{rw} = (\{\bot_{\mathbb{E}}, \mathsf{rd}, \mathsf{wr}\}, \sqcup, \sqcup, \bot_{\mathbb{E}})$ is an effect quantale. The effect ordering is $\bot_{\mathbb{E}} \sqsubseteq \mathsf{rd} \sqsubseteq \mathsf{wr}$, and $\sqcup$ is the corresponding least upper bound. The bottom element $\bot_{\mathbb{E}}$ denotes the no-op effect and identity element. It is also straightforward to show the definition satisfies the distributivity law, as $\rhd$ is equivalent to $\sqcup$.

$$\textsc{E-deref}\ \frac{\Gamma \mid \Sigma \vdash t : (\mathsf{Ref}\ T)^q \mid \epsilon}{\Gamma \mid \Sigma \vdash !t : T^\perp \mid \epsilon \rhd \{(q^*, \mathsf{rd})\}} \qquad \textsc{E-assign}\ \frac{\Gamma \mid \Sigma \vdash t_1 : (\mathsf{Ref}\ T)^q \mid \epsilon_1 \qquad \Gamma \mid \Sigma \vdash t_2 : T^\perp \mid \epsilon_2}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \mathsf{Unit}^\perp \mid \epsilon_1 \rhd \epsilon_2 \rhd \{(q^*, \mathsf{wr})\}}$$

Fig. 6. The typing rules for dereference and assignment with read/write effect.

In the above definition, the ordering among no-op, read, and write effects is a chain. Other ways to structure effects and ordering are also possible. For example, $\mathcal{P}(\{\mathsf{rd}, \mathsf{wr}\})$ ordered by set-inclusion provides a finer-grained representation and allows to control read and write effects separately.

Next, the generic framework is instantiated with $\Delta(\mathbb{E}_{rw})$, where $\mathbb{E}_{rw}$ is the codomain of the store-sensitive effect quantale. Then we can define proper typings for dereferencing and assignments (Fig. 6). The instantiation is flow-insensitive, because $\mathbb{E}_{rw}$ does not distinguish join and sequential composition. Thus the join and sequential composition of store-sensitive effect quantales are equivalent too. The type-and-effect rules are straightforward: dereferencing terms $!t$ induces the read effect rd over α^*. In the conclusion of the assignment typing rule (E-Assign), the inherited effects are sequentially composed with the write effect wr over the aliases of the target reference. The latter effect is called the *intrinsic effect* [Lucassen and Gifford 1988] of this term.

Soundness. To establish soundness, we give the dynamic semantics of dereference and assignment in the form with effect instrumentation:

$$\begin{array}{llll} C[!l] \mid \sigma & \rightarrow^{\{(l,\mathsf{rd})\}} & C[\sigma(l)] \mid \sigma & l \in dom(\sigma) \\ C[l := v] \mid \sigma & \rightarrow^{\{(l,\mathsf{wr})\}} & C[\mathsf{unit}] \mid \sigma[l \mapsto v] & l \in dom(\sigma) \end{array}$$

The soundness of this instantiation naturally follows (Proposition 4.4), in the sense that the runtime read/write effects are captured by our store-sensitive effect representation.

4.3 Tracking Destructive Effects with Flow-Sensitive Instantiation

In Section 3, the T-Ref rule restricts reference cells so that they can only store values of untracked type, *i.e.*, t must have type $T^\perp$ for ref t. In this section, we extend our language with nested mutable references and move semantics that relax this restriction. We first motivate flow-sensitive effects and move semantics with informal examples, before discussing their typing rules (Fig. 7) and dynamic semantics (Section 4.3.3).

4.3.1 Move Semantics and Ownership. The reference type is generalized by adding type Ref $T^\varnothing$. Values of type Ref $T^\varnothing$ are references for *tracked* values, in contrast to values of type Ref T, whose referents are *untracked* values. In the generalized reference type, the qualifier of the inner type must be the empty set, indicating that the cell uniquely holds the value without any other aliases. As before, we permit dereferencing ($!t$) and assignment ($t_1 := t_2$) only for untracked references. To handle tracked references, we introduce the new syntax forms move and swap.

Values of Ref $T^\varnothing$ are created using the same overloaded syntax ref t. Creating a tracked reference induces the move semantics — the reference cell becomes the unique owner of t, and disables accesses via other aliases of t afterwards. One can also explicitly transfer ownership by using the new construct move t, which consumes t by disallowing accesses of t's aliases afterwards. The returned value of move is a fresh, non-aliased, but tracked value. Therefore when move t is bound to a variable, that variable becomes the unique owner of the value computed by t. To illustrate the effect of move, consider the following program:

```
val x = new Ref(1); val y = x // x : Ref[Int]{x} and y : Ref[Int]{x,y}
val z = move(x)               // z : Ref[Int]{z}; x and y are killed
!x + !y                       // both !x and !y yield type errors at this point
```

$$T ::= \cdots \mid \mathsf{Ref}\ T^{\varnothing} \quad t ::= \cdots \mid \mathsf{swap}\ t_1\ t_2 \mid \mathsf{move}\ t \qquad C ::= \cdots \mid \mathsf{swap}\ C\ t \mid \mathsf{move}\ C \mid \mathsf{swap}\ v\ C$$

$$\frac{\Gamma \mid \Sigma \vdash t : T^{\alpha} \mid \epsilon_1}{\Gamma \mid \Sigma \vdash \mathsf{ref}\ t : (\mathsf{Ref}\ T^{\varnothing})^{\varnothing} \mid \epsilon_1 \rhd \{(\alpha^*, \mathsf{kill})\}}\ \mathrm{E}_k\text{-REF} \qquad \frac{\Gamma \mid \Sigma \vdash t : (\mathsf{Ref}\ T^{q})^{\alpha} \mid \epsilon_1}{\Gamma \mid \Sigma \vdash \mathsf{move}\ t : (\mathsf{Ref}\ T^{q})^{\varnothing} \mid \epsilon_1 \rhd \{(\alpha^*, \mathsf{kill})\}}\ \text{E-MOVE}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : (\mathsf{Ref}\ T^{\varnothing})^{\alpha} \mid \epsilon_1 \qquad \Gamma \mid \Sigma \vdash t_2 : T^{\beta} \mid \epsilon_2 \qquad \beta^* \cap \alpha^* = \varnothing}{\Gamma \mid \Sigma \vdash \mathsf{swap}\ t_1\ t_2 : T^{\varnothing} \mid \epsilon_1 \rhd \epsilon_2 \rhd \{(\beta^*, \mathsf{kill}), (\alpha^*, \mathsf{wr})\}}\ \text{E-SWAP}$$

Fig. 7. The syntax and new typing rules for flow-sensitive constructs.

In this program, x is a reference to an untracked value, and is reachable by y. However, moving x to z eliminates all existing aliases of x. Therefore, after the move operation, variable z becomes the unique owner that can access the referent of x, and variables x and y become inaccessible.

The behavior of swap $t_1\ t_2$ is to *move* the value of t_2 into the cell pointed by t_1, and to return the old value from the cell of t_1. The returned value of swap is a fresh, non-aliased, but tracked value. Meanwhile, by moving the value of t_2 into that cell, all aliases of t_2 are killed afterwards. Since t_1 must be a value of $\mathsf{Ref}\ T^{\varnothing}$ and T could also be a reference type, one may use swap to manipulate nested references. In those cases, t_1 is a nested reference and t_2 is a flat reference. To illustrate the semantics, consider the following example:

```
val x  = new Ref(1); val y = new Ref(2)  // x  : Ref[Int]{x} and y : Ref[Int]{y}
val nc = new Ref(y)                      // nc : Ref[Ref[Int]∅]{nc}; y is killed
val z  = swap(nc, x)                     // z  : Ref[Int]{z}; x is killed
```

The example above first allocates two cells of integers, and then creates a nested cell nc that contains the reference y. The creation of nc also kills the prior aliases of y, including y itself. By swapping nc with x, the referent of nc (*i.e.*, the reference pointing to value 2) is retrieved and bound to z; meanwhile, the reference x is moved into nc. After the swap, both x and y are not usable anymore.

Moving or swapping a function with non-empty qualifiers is also allowed, which causes the variables captured by the function to be killed as well.

4.3.2 Flow-Sensitive Instantiation. The move/swap operations induce effects that can invalidate any subsequent operations in the control flow. Accordingly, we extend the effect labels with destructive effects. Although it is possible to have a finer distinction between destructive effects (*e.g.*, distinguish ownership transfer and deallocation), we show the simplest model that only adds one dominant kill effect. It invalidates all other operations, including itself, and is enough for our language extension. The definition of the extended effect labels is shown below:

Table 1. $\rhd$ operator for Definition 4.6.

$\rhd$	$\bot_{\mathbb{E}}$	rd	wr	kill
$\bot_{\mathbb{E}}$	$\bot_{\mathbb{E}}$	rd	wr	kill
rd	rd	rd	wr	kill
wr	wr	wr	wr	kill
kill	undefined			

Definition 4.6 (Effect Labels with kill). The extended effect labels are still defined as an effect quantale $\mathbb{E}_k = (\{\bot_{\mathbb{E}}, \mathsf{rd}, \mathsf{wr}, \mathsf{kill}\}, \sqcup, \rhd, \bot_{\mathbb{E}})$. The kill effect is the top element of the ordering, *i.e.*, $\bot_{\mathbb{E}} \sqsubseteq \mathsf{rd} \sqsubseteq \mathsf{wr} \sqsubseteq \mathsf{kill}$, and $\sqcup$ computes the least upper bound following this ordering. The new sequential composition operator $\rhd$ is defined in Table 1, where the first column enumerates the left-hand side operand, and the first row the right-hand side operand.

It is worth noting that the sequential composition operator $\rhd$ is *partial* after introducing the kill effect. If the left-hand side operand of $\rhd$ is kill, then $\mathsf{kill} \rhd e$ is undefined for any $e \in \mathbb{E}_k$. This corresponds to those cases that mention or use a variable after it was killed, which is illegal, yielding a type error. We now show that $\mathbb{E}_k$ is indeed an effect quantale with distributivity:

Lemma 4.7. *With respect to Definition 4.6, the sequential composition operator $\triangleright$ distributes over join $\sqcup$ in both directions, i.e., $a \triangleright (b \sqcup c) = (a \triangleright b) \sqcup (a \triangleright c)$ and $(a \sqcup b) \triangleright c = (a \triangleright c) \sqcup (b \triangleright c)$.*

By instantiating the store-sensitive effect quantale with $\mathbb{E}_k$, we can give the type-and-effect rules for the new allocation primitive, move, and swap (Fig. 7). The E_k-ref rule states that an allocation consumes/kills all the aliases (denoted by α^*) that can be used to access the resources. The premise of E_k-ref requires that t is a tracked value, where the untracked case is handled by the old rule (cf. T-ref, Fig. 3). In the flow-sensitive variant, using the transitive lookup (*e.g.*, α^*) is necessary to ensure safety, because a potential destructive effect (such as kill) over an identifier affects all the variables it points to. The E-move rule kills existing aliases of a reference term, resulting in a fresh tracked reference. The inner type of the reference can be either tracked or untracked, therefore t is typed as Ref T^q. The E-swap rule only allows swap operations between disjoint parts of the store, *i.e.*, $\beta^* \cap \alpha^* = \emptyset$. When constructing the intrinsic effect of a swap (*i.e.*, kill over β^* and wr over α^*), the disjointness side-condition also ensures that the resulting store-sensitive effect is well-formed.

4.3.3 Soundness. To establish the soundness given by the framework, we give the effect-instrumented dynamic semantics for the flow-sensitive constructs.

$$
\begin{array}{llll}
C[\mathsf{ref}\ v] \mid \sigma & \rightarrow^{\{(FL(v),\mathsf{kill})\}} & C[l] \mid (\sigma, l \mapsto v) & l \notin dom(\sigma) \\
C[\mathsf{move}\ l] \mid \sigma & \rightarrow^{\{(l,\mathsf{kill})\}} & C[l'] \mid \sigma' & (\sigma', l') = transfer(\sigma, l) \\
C[\mathsf{swap}\ l\ v] \mid \sigma & \rightarrow^{\{(FL(v),\mathsf{kill}),(l,\mathsf{wr})\}} & C[v'] \mid \sigma'[l \mapsto v] & (\sigma', v') = transfer(\sigma, \sigma(l))
\end{array}
$$

The static move semantics allows reference cells to be nested or to store function values, and it induces kill effects over a set of aliased variables. The dynamic semantics reflects this behavior by inducing kill effects over store locations. To define such semantics, we use two auxiliary functions:

- Given a value v, $FL(v)$ syntactically computes its set of *free locations*, *e.g.*, $FL(l) = \{l\}$ for location values l, and similarly $FL(\lambda f(x).!l) = \{l\}$ for functions. In the reduction rules for ref and swap, we use $FL(v)$ as the target of kill effects.
- To model ownership transfer at runtime, we use the function $transfer(\sigma, v)$, whose argument is a store σ and a potentially *killed* value v. The function returns a store and a value corresponding to σ and v that can be safely used without worrying if they are alive. One way to implement the function is by copying relevant cells to freshly generated locations. For example, $transfer(\sigma, l)$ would return a fresh location l' pointing to the same value of $\sigma(l)$, as used in the rule for move. Similarly, the rule for swap uses *transfer* to "freshen" the old value at location l.
  In a dynamic semantics that is closer to actual implementations, we can erase the instrumentation and avoid the copying, since the effect preservation theorem already proves that well-typed programs will not perform operations over killed variables. Therefore, we can safely reuse the same value/location and store to fulfill the *transfer* function.

Finally, the effect preservation property (Proposition 4.4) holds for the flow-sensitive effect system presented here, analogous to the flow-insensitive version.

4.4 Discussion and Extensions

"Gen" Effects via Strict Must-Reachability. It is useful to extend the flow-sensitive effect system with tracking of resource initialization as an application of "gen" effects. In contrast to kill effects that disable any further use, certain operations on resources are valid only after they have been properly initialized. This extension can be built on top of must- and strict-reachability (Section 3.9) and yields flow-sensitive "must-init" effects. Strict vs. non-strict reachability provides the necessary means to track finer-grained object initialization effects [Kabir et al. 2020; Liu et al. 2020; Summers and Müller 2011], *e.g.*, distinguishing if the object skeleton has been initialized but not its fields.

Use vs Mention. Recall that the flow-sensitive sequential composition operator $\triangleright$ (Definition 4.6) prohibits any "mention" or "use" of killed variables. This choice effectively prevents erroneous programming patterns, but also some arguably benign programs, for example, defining but not invoking a function that closes over a killed variable:

```
val x = new Ref(0); val y = move(x) /* move kills x */
def f() = { x /* current system prevents mentioning x, since it x is killed */ }
```

This program is still safe as long as callers of `f()` do not "use" the returned value. In fact, to establish soundness, preventing "use-after-kill" is sufficient. We may regain expressiveness by carefully allowing limited forms of "mention-after-kill" (*e.g.*, by defining $\mathsf{kill} \triangleright \bot_{\mathbb{E}}$ as kill) while disallowing any "use-after-kill". This relaxation can enable certain useful programming patterns. For example, in a capability system, we may need to pass (*i.e.*, mention) a capability on one side of an abstraction boundary, indifferent to the fact that it has been killed on the other side. However, extra caution is required as blindly allowing "mention-after-kill" may violate effect safety. For example, consider the following function and its invocation:

```
// : () => Ref[Int]^∅, latent effect: {(∅, kill)}
def g() = { val x = new Ref(0); val y = move(x); x }
val c = g() // : Ref[Int]^{c}, but c in fact has been killed
```

The function `g` returns a dangling local reference `x`, but clients will not be able to distinguish it from a freshly allocated reference, since `x` is not in scope and is substituted with the empty set in `g`'s latent effect. To prevent such erroneous programs, we can either (1) add ad-hoc checks to examine if a killed value is returned, (2) extend the base λ^* system so that the qualifiers also track an "effect status" (*e.g.*, is-killed, is-initialized, etc.) for each variable, at the cost of complicating the base system, or (3) use self-types to type allocations so that escaping references are still named and explicitly tracked in the latent effect. With this option, the type of function `g` would be $() \to \mu z.\{(\mathsf{Ref}\ T)^z \mid (z, \mathsf{kill})\}$, introducting a self reference `z` in the return type to enable `g`'s latent kill effect to refer to the value being returned. Instead of being placed on the function arrow as a property of a function, in this model a function's latent effect should be thought of as characterizing the *computation* of the function result, and hence being part of the function's result type. It is not necessary to track every effect (*e.g.*, read/write) on return values of functions in this way, but only flow-sensitive effects that enable/disable subsequent operations at the call-site (*i.e.*, gen/kill).

5 CASE STUDIES AND EXAMPLES

This section presents several applications of the λ^* and λ^*_ϵ type systems. We first show that λ^* can encode capabilities for (algebraic) effects [Brachthäuser et al. 2020a; Osvald et al. 2016], where effect types express context requirements of computations with (user-defined) effects. Then we show that λ^*_ϵ can be extended to express one-shot continuations and race-free parallel computations.

5.1 Algebraic Effects and Handlers

The type system of λ^* readily supports non-escaping function arguments as an emergent property of tracking reachability (Section 2.4.3). We may leverage these for second-class capabilities [Osvald et al. 2016] as a way to integrate side effects into a programming language. The `try`/`catch`-style delimiter functions neatly generalize to algebraic effect handlers [Plotkin and Power 2003; Plotkin and Pretnar 2009], and "capability-passing style" informed new language designs in this area [Brachthäuser et al. 2020a,b; Schuster et al. 2020]. In this section, we consider examples from Effekt [Brachthäuser et al. 2020a]. The key take-away is that λ^* may serve as a flexible IR for languages with effect handlers, supporting multiple points in their design space.

5.1.1 Effects as Capabilities. Algebraic effects declare nominal effect interfaces, *e.g.*,

```
effect Fail(msg: String): Nothing // signal failure
```

```
case object Fail, Next, Choice
type CanFail   = (Fail⊥   => String⊥ => Nothing⊥)∅
type CanNext   = (Next⊥   => Unit⊥   => String⊥ )∅
type CanChoice = (Choice⊥ => Unit⊥   => Boolean⊥)∅
// : (stringToInt(String⊥) => (doFail: CanFail) => Int⊥)∅
def stringToInt(str, doFail) = toInt(str) match case Some(n) => n; case None => doFail(Fail)("error")
// : (number(doNext: CanNext) => (doFail : CanFail) => Int⊥)∅
def number(doNext, doFail) = stringToInt(doNext(Next)(), doFail)
// : (many(block: (() => Unit⊥)ᵠ) => (doChoice: CanChoice) => Unit⊥)ᵠ+∅
def many(block, doChoice) = while (doChoice(Choice)()) block()
```

Fig. 8. Programming and composing computations with multiple user-defined effects in capability-passing style (examples adapted from [Brachthäuser et al. 2020a]).

```
def HChoice : (CanChoice => A∅)∅ => (A∅ => B∅)∅ => (() => (Bool⊥ => B∅)∅ => B∅) => B∅
def choose(doChoose, a, b) = { if (doChoose(Choice)()) a else b }
val res = HChoice { c  => choose(c,1,2) * choose(c,3,4) } { x => List(x) }
                  { () => resume => resume(true) ++ resume(false) } // yields [3,4,6,8]
```

Fig. 9. Generic signature of an effect handler `HChoice` for the `Choice` effect with usage example.

which introduce effect types such as `Fail`, and associated operations[2] to trigger the corresponding effect, *e.g.*, `Fail` takes a string message and returns the empty type `Nothing`, since it is not supposed to return. We encode the effect operations in λ^* as capabilities with analogous signatures

```
case object Fail;  type CanFail = (Fail⊥ => String⊥ => Nothing⊥)∅
```

where we tag the operations with singletons (additional base types) as a lightweight encoding of nominality. We do not track an effect capabilities' arguments, but the capability itself with the empty qualifier ∅. This prevents assignments to mutable references, emulating an aspect of second-class values. Programs/functions that induce effects either bind a respective capability value in their parameter list, provided by the calling context, or they may close over a specific capability at their definition site. Composing effectful programs accumulates the capability requirements of the sub-programs, as illustrated in Fig. 8.

5.1.2 Lightweight Effect Polymorphism. The capability-passing style yields a lightweight effect polymorphism [Brachthäuser et al. 2020a; Osvald et al. 2016] with a less verbose effect type syntax than languages with explicit effect polymorphism (*e.g.*, Koka [Leijen 2017] and Helium [Biernacki et al. 2019]). For instance, this higher-order function from [Brachthäuser et al. 2020a]

```
def eachLine[A](file: File⊥, f: (String⊥ => A⊥)∅): Unit⊥
```

is implicitly effect-polymorphic because the argument `f` may close over arbitrary effect capabilities:

```
eachLine(file, { s => if doChoice() doPrint(s) else doFail("failure") })
eachLine(file, { s => doNext() ++ s })
```

yet the signature mentions no explicit effect types. Lexically closing over specific effect capabilities prevents that `eachLine` accidentally handles the effects of `f`, *i.e.*, it is effect parametric [Biernacki et al. 2020; Zhang and Myers 2019]. Varying the signature indicates a different semantics, *e.g.*,

```
def eachLine[A](file: File⊥, f: (String⊥ => CanFail => A⊥)∅): Unit⊥
```

where `f` takes a `CanFail` capability. Brachthäuser et al. [2020a]'s language guarantees that `eachLine`'s implementation provides a capability for the `Fail` effect. Any other effect of `f` "bubbles" into the calling context. Similarly, our `eachLine` encoding has an empty qualifier, prohibiting closure over an external capability, so that its implementation necessarily has to call `f` with a fresh one.

5.1.3 Preventing Capability Leaks. Capabilities should not escape their lexically delimiting effect handlers. For instance, neither should `throw` (Section 2.4.3) escape its `try` block, nor should

[2]For simplicity, and without loss of generality, we associate exactly one effect operation with each effect type, while Effekt and other algebraic effect languages permit grouping multiple operations under one effect type.

`eachLine`'s internal `CanFail` capability escape it. We can easily determine escaping parameters and thus enforce well-scopedness via the qualifiers of the function codomains, *e.g.*,

```
// : (f(doFail: CanFail) => g(() => Nothing^⊥)^{doFail})^∅
def f(doFail) = { () => doFail(Fail)("Escaped!") }
```

Escaping by assignment to a reference cell is also prohibited by the qualifier of a capability:

```
def f(doFail) = { x := doFail } // error: qualifier mismatch ∅ ≠ ⊥.
```

Brachthäuser et al. [2020a] prevent escaping capabilities by only allowing second-class functions/blocks in the language. Their restriction prohibits some effect handlers, *e.g.*, functional state. We permit first-class closures, recovering the lost expressiveness.

5.1.4 Effect Handlers. Handlers generalize the `try` example from (Section 2.4.3), *e.g.*, Fig. 9, shows the generic signature of a handler combinator `HChoice` for the `Choice` effect. Handlers transform computations of type `A` having some effect `E` (*e.g.*, `Choice`) to computations of type `B`. Their action is determined by (1) a return clause `onRet` in case the `block` yields an answer value, and (2) effect clauses handling the effect type's operations when triggered by `block`. These mirror the type signature of the operation, and expose the resumption/continuation of the underlying computation.

A handler application supplies a unique capability for the effect to the given block. In this way, blocks may unambiguously distinguish between multiple instances of the same effect when nesting handlers [Biernacki et al. 2020]. Operationally, handlers need to capture control flow for the resumptions. These may be implemented in a number of ways, *e.g.*, using control operators or monads [Brachthäuser et al. 2020b], or variants of continuation-passing style (CPS) [Cong et al. 2019; Hillerström et al. 2020, 2017; Kammar et al. 2013; Rompf et al. 2009; Schuster et al. 2020], which we can in principle implement in λ^* (cf. Section 5.2).

Finally, we support different kinds of handlers and refinements, which we may freely vary: (1) Standard *deep handlers* [Kammar et al. 2013] (as shown in Fig. 9), and *shallow handlers* [Hillerström and Lindley 2018], by changing the return type of `resume` from `B` to `A`. (2) Generalizing to λ^*_ϵ, we obtain *one-shot resumptions* by our construction from Section 5.2. (3) We may selectively enforce *non-escaping resumptions*, enabling further optimization opportunities in a CPS implementation [Cong et al. 2019]. We conjecture that our encodings also permit a statically-checked version of the related notion of scoped resumptions [Xie et al. 2020], which we leave for future work.

5.2 One-Shot Continuations

Control operators, *e.g.*, call-with-current-continuation and shift/reset [Danvy and Filinski 1990; Felleisen 1988], provide powerful control abstractions. Here, we consider variants of control operators reifying affine ("one-shot") continuations, which can be invoked at most once. These continuations can be implemented efficiently [Bruggeman et al. 1996], and are supported by mainstream functional programming languages [Cisco Systems, Inc. 2017; Sivaramakrishnan et al. 2021].

The following examples demonstrate legal and illegal uses of a one-shot delimited continuation `k`, captured by `shift`^{1s}, the affine variant of `shift`:

```
reset { 1 + shift^1s { k => k(2) + 3 } }     // okay,  k is invoked once, result is 6
reset { 1 + shift^1s { k => 3 } }            // okay,  k is not invoked,  result is 3
reset { 1 + shift^1s { k => k(2) + k(3) } } // error, k is invoked twice
```

However, existing implementations cannot statically enforce that `k` is affine. Instead, violations trigger runtime errors when `k` is invoked more than once.

It is straightforward to type a family of control operators C with different static properties in λ^*_ϵ. Fig. 10 (left) shows the generic typing rule (T-CTRL) which is for the most part standard [Duba et al. 1991]. We include three additional parameters which may freely vary to obtain different versions of control operators: (1) *Ret* is the return type of the reified continuation k, (2) *KE* determines whether

T-CTRL

$$\frac{\Gamma, k : (k(x : T^{q_1}) \xrightarrow{\epsilon \triangleright KE} Ret)^k \vdash t : T^{q_2} \mid \epsilon_1 \quad NE}{\Gamma \vdash C\ k\ \text{in}\ t : T^{q_1} \mid \epsilon_1}$$

$$\begin{array}{l} \text{let/cc}^{1s} : Ret = \text{Nothing}^{\perp}, KE = \{(k, \text{kill})\}, NE = true \\ \text{shift}^{1s}_{ne} : Ret = S^{q_3}, KE = \{(k, \text{kill})\}, NE = k \notin FV(T^{q_2}) \end{array}$$

Fig. 10. Left: A generic typing rule for a family of control operators. Right: Instantiations that are one-shot control operators allowing (let/cc^{1s}) and disallowing (shift$^{1s}_{ne}$) escaping, respectively.

k should be one-shot, and (3) NE determines whether k should be non-escaping. In Fig. 10 (right), we show two representative instantiations:

(1) let/cc^{1s} captures one-shot undelimited continuations, where the return type of continuation k is Nothing$^{\perp}$ that can be up-cast to any other type by subsumption. We enforce the one-shot property by requiring KE to be $\{(k, \text{kill})\}$, which disallows any use of k after its first invocation. To allow multi-shot continuations, we let KE be the empty set. The effect KE is sequentially composed with δ, which is the latent effect of the rest of the computation, depending on the evaluation context.

(2) shift$^{1s}_{ne}$ captures one-shot, non-escaping, delimited continuations, where the return type of the continuation k is generalized to an arbitrary type. In addition to enforcing that k is one-shot, we also prevent it from escaping by requiring $k \notin FV(T^{q_2})$ (see Section 2.4.3 and Section 5.1.3).

In the presence of nested mutable state (Section 4.3), the rule T-CTRL does not prevent the continuation k from escaping by assignment. Consider an example that stores the reified continuation in a cell allocated in the outer scope by using the swap operation. The swap operation induces a kill effect over k. However, our typing rule also imposes the kill effect over k once called, so we cannot distinguish whether the continuation is killed by an escaping assignment or killed by its own one-shot property. If the continuation is killed by an escaping assignment, it can still be invoked from some other place, violating the one-shot property. However, we can resolve this issue by refining the effect labels to introduce and distinguish two different causes of kill effects, then requiring one and excluding the other one.

The typing rules and effect quantale of λ^*_ϵ can be extended to directly express the control effect as well [Gordon 2020b]. Tracking answer type modification [Danvy and Filinski 1989] using the sequential type-and-effect system is also possible, but orthogonal to this case study.

5.3 Race-Free Parallel Computation

Non-interference between two terms [Reynolds 1978] allows their safe parallel execution. In Section 2.4.2, we showed that requiring disjoint reachability qualifiers ensures non-interference. However, this requirement is too coarse-grained, *e.g.*, it excludes terms with shared read-only references, since these will show up in their qualifiers. Here, we utilize the effect system of λ^*_ϵ (Section 4) for more fine-grained distinctions, allowing reads but excluding writes to shared data. Our approach is inspired by Rust's "shared XOR mutable" principle [Jung et al. 2021]. Consider the following two programs, each aliasing `c1` and `c2`:

```
val c1 = new Ref(0); val c2 = c1
par({ !c1 }, { !c2 }) // no interference
// ok: parallel rd on aliased c1 and c2
```

```
val c1 = new Ref(0); val c2 = c1
par({ c1 := 1 }, { !c2 }) // interference
// error: parallel wr/rd on aliased c1 and c2
```

In the λ^*_ϵ effect system, the program to the left is safe, as the shared references are only read in both threads. The program to the right will be rejected, as the threads both read and write the shared reference `c1`. We encode the intuition that two threads can have at most read effects over shared references by a type-and-effect rule for a general parallel combinator parPair in λ^*_ϵ:

$$\frac{\Gamma \vdash t_1 : T_1^{q_1} \mid \epsilon_1 \qquad \Gamma \vdash t_2 : T_2^{q_2} \mid \epsilon_2 \qquad \text{non-interfering}(\epsilon_1, \epsilon_2)}{\Gamma \vdash \text{parPair}(t_1, t_2) : (T_1^{q_1}, T_2^{q_2})^{q_3} \mid \epsilon_1 \sqcup \epsilon_2}\ \text{E-PAR-PAIR}$$

The typing rule E-PAR-PAIR is mostly standard, while the relevant part to parallel execution is the side-condition over effects ϵ_1 and ϵ_2 to enforce non-interference:

$$\text{non-interfering}(\epsilon_1, \epsilon_2) \equiv \forall(\alpha_1, e_1) \in \epsilon_1, \forall(\alpha_2, e_2) \in \epsilon_2, \alpha_1 \cap \alpha_2 \neq \varnothing \Rightarrow (e_1 \sqcup_{\mathbb{E}} e_2) \sqsubseteq \mathsf{rd}$$

This condition intuitively means that if the two computations t_1, t_2 have any shared aliasing, then their joined effects $e_1 \sqcup_{\mathbb{E}} e_2$ are at most the read effect, which is safe for parallel execution. Our parPair construct is compositional, and can encode diverse concurrency abstractions, *e.g.*, the `join` operator in Rust [Jung et al. 2021].

6 RELATED WORK

Type Systems for Second-Class Values. Osvald et al. [2016] use type qualifiers to distinguish 1st- and 2nd-class values. While our qualifiers list all reachable variables, theirs denote an upper bound on the "classyness" or *privilege level* of reachable values: *e.g.*, closures over a 2nd-class value are also 2nd-class. Their privilege qualifiers can be mapped to regular types, enabling linguistic reuse of host-language type abstraction features and a fine-grained privilege lattice, which can, *e.g.*, mediate access to different kinds of capabilities (*e.g.*, read vs. write) in different ways, based on their types. Osvald et al. [2016] prove a strong soundness theorem that guarantees stack-bounded lifetimes for non-1st-class values. This requires a conservative formulation of their calculus that prohibits functions from returning non-1st-class values. It is known that this restriction is not essential for weaker notions of soundness, such as syntactic preservation of privilege qualifiers during reduction, which is still sufficient to express non-escaping properties for scoped introduction forms. In comparison, our soundness result ensures strong memory properties (*i.e.*, separation of terms with disjoint reachable sets) without broad restrictions and is also able to express properties such as non-escaping (cf. examples in Section 2.3). It would be interesting to integrate aspects of both systems, *e.g.*, deriving stack allocation policies from reachability types, or complementing reachability types with type-based privilege levels as an additional classification mechanism. While we can broadly identify untracked and tracked values with Osvald et al. [2016]'s 1st- and 2nd-class values, respectively, our system does not currently model finer-grained privilege levels.

Substructural Type Systems. Linear type systems [Turner et al. 1995; Wadler 1990] dictate that functions use their arguments exactly once, while affine type systems permit at-most-once use. Linear Haskell [Bernardy et al. 2018] uses multiplicity to track how often a function may use its argument based on how often its result is consumed by a caller. Dual to linear types, uniqueness types [Barendsen and Smetsers 1996; de Vries et al. 2006, 2007] ensure that the argument of a function is unique within the function's context, even if the callee can work with the argument as it pleases. Reachability types in λ^* alone do not restrict the number of variable uses, but can express that arguments and functions are separate or overlapping, which is conceptually different from linear type systems. Our λ^*_ϵ system uses a flow-sensitive effect system to enforce flavors of affine uses and uniqueness (Section 4).

We treat functions similar to bunched typing (BT) [O'Hearn 2003] and syntactic control of interference [O'Hearn et al. 1999; Reynolds 1978, 1989], where the $A \mathrel{-\!\!*} B$ arrow represents functions with resources disjoint from their arguments. However, we do not enforce fully disjoint resources to evaluate the function and argument *expressions* in applications. Whereas BT demands a split environment as for linear types, our work reasons about storage separation between the computed function and argument *values*. BT also does not model mutable state. Note that we *can also* achieve full non-interference of computations, if desired, by requiring two *functions* to be separate.

Alias types [Smith et al. 2000] and L^3 [Ahmed et al. 2007] define type systems for low-level languages. L^3 supports strong updates, in the sense that a mutable cell can store values of different types over time. Supporting strong updates is not a goal of our work; we aim to track shared data

in higher-order functional languages. Both alias types and L^3 use existential types for tracking escaping locations, which comes at a cost of readability and additional term-level footprint from explicit introduction and elimination forms. For comparison, consider our `returnFresh` function (Section 2.2.2), whose type signature succinctly expresses that the returned reference is fresh, without any quantifiers. Furthermore, the escaping closures example (Section 2.3) uses the function's self reference as a form of implicit quantifier. Even though an "existential" type was assigned, the result is still a function and immediately usable as such without term-level unwrapping.

Ownership Types. Ownership type systems [Clarke et al. 2013; Noble et al. 1998] enforce unique access paths, and can structure heaps in various ways, such as object contexts [Clarke et al. 2001, 1998], trees [Boyapati et al. 2002; Clarke et al. 2008], nested regions [Zhao et al. 2008], universes [Müller and Poetzsch-Heffter 2000], ownership topology [Dietl et al. 2011], and islands of objects [Noble et al. 1998]. Hogg [1991] and Naden et al. [2012] re-introduce selective sharing via borrowing. Clebsch et al. [2015a] use reference counting to allow sharing.

Inspired by the concept of ownership and borrowing, the Rust programming language [Klabnik and Nichols 2019; Matsakis and Klock 2014] adopts a strong ownership-based type system that requires a memory location to be either uniquely owned by a mutable reference, or allows it to be shared among multiple read-only references. Similar models are explored by other programming language communities, *e.g.*, Swift [The Swift Developer Community 2019] and D [Bright 2019].

Our reachability type system and these traditional ownership systems start from two ends of a spectrum: ownership systems start from the uniqueness restriction and then selectively re-introduce sharing in a controlled manner (*e.g.*, Rust); in contrast, λ^* allows sharing and checks separation in the first place, then layers additional mechanisms on top for selectively enforcing uniqueness.

Tracking Captured Variables. In parallel to our work, Boruch-Gruszecki et al. [2021] proposed a type system to track variables captured by closures. Like ours, their system also uses sets of variables as type qualifiers, but it does not appear to impose similar separation boundaries between function arguments and free variables. We look forward to seeing how this work matures on its way to formal publication. Similarly, Scherer and Hoffmann [2013] track data-flow from the environment into function closures, by annotating function types with type contexts. In contrast, our work tracks reachable values through reachability qualifiers.

Regions. Tofte and Talpin [1997] use lexically-scoped expressions to delimit the lifetime of regions. Gay and Aiken [1998] use reference counting to track live references in a region, which increases precision and leads to more efficient memory use through early deallocation. Cyclone [Grossman et al. 2002] goes beyond purely lexical lifetimes by using existential types to express escaping closures. In contrast, λ^* uses function self-references to express escaping closures and references. Lu and Potter [2005] use regions in types to specify reachability relations between objects, and to prevent unwanted reference cycles. Instead of imposing a region discipline, λ^* uses type qualifiers to track which other values are reachable from a given expression's result.

Walker et al. [2000] use a type system to track non-aliasing (uniqueness). Instead of using lexical scope to control the lifetime of regions, Walker et al. [2000] use static capabilities to check the availability of regions before accessing them. The "alias calculus" [Kogtenkov et al. 2015] aims to prove functional correctness of OO programs by inferring values unchanged by a given method, which could potentially be achieved using our λ^*_ϵ system (Section 4), but has not been our goal.

Type and Effect Systems. have been proposed to check or infer local non-aliasing [Aiken et al. 2003], and to track read and write effects on the heap [Gifford and Lucassen 1986; Lucassen and Gifford 1988; Nielson and Nielson 1999], exceptions [Gifford and Lucassen 1986], purity [Pearce 2011], atomicity [Abadi et al. 2008], locks [Gordon et al. 2012] and others. Clarke and Drossopoulou

[2002] extend ownership types with effects to analyze non-interference in OO programs. In our reachability type-and-effect system λ^*_ϵ, the store-sensitive quantale (Section 4) is a novel instance of the effect quantale framework introduced by Gordon [2021]. Ivaskovic et al. [2020] relax the distributivity requirement of effect quantales so that data-flow analyses can be uniformly encoded as effect systems. Gordon [2020b] discusses modeling control operators, *e.g.*, call/cc, using sequential effect systems, which is related to our case study in Section 5.2. Typestate-based systems [Aldrich et al. 2009; DeLine and Fähndrich 2001] rely on a flow-sensitive must-analysis. It is possible to extend our system with must-reachability and must-effects (see Section 3.9 and Section 4.4).

Capabilities and Permissions. Capability systems [Boyland et al. 2001; Castegren and Wrigstad 2016; Steed and Drossopoulou 2016] have been used to reason about program resources. Haller and Odersky [2010] use capabilities to ensure unique access with borrowing. Fractional permissions [Boyland 2003] provide more fine-grained control, *e.g.*, enforcing unique mutation and allowing shared read accesses. The work of Yasuoka and Terauchi [2009] associates regions with fractional capabilities to reason about program resources, *e.g.*, reclaimable memory regions. Instead of specifying which references can be accessed, Clebsch et al. [2015b] use capabilities to demand what other aliases are not allowed. Gordon [2020a] articulates the use-mention distinction when designing capabilities and effect systems. Inspired by that, we treat "use" and "mention" differently. Moreover, our type system goes beyond the use/mention classification, and tracks aliases, "kill" effects, and (potentially) "gen" effects, which are not considered by Gordon [2020a].

Program Logics. Separation logic (SL) [Ishtiaq and O'Hearn 2001; O'Hearn et al. 2001; Reynolds 2002] extends Hoare logic with separating conjunctions to express that two memory locations are disjoint. SL has proven especially effective for verifying low-level systems software. Krishnaswami [2006] extends SL with an equational theory to reason about closures in a typed higher-order language. Recently, Charguéraud [2020] formalized SL for a λ-calculus with imperative features in Coq. Our type system is inspired by SL, *e.g.*, function types may specify that a given argument does not overlap with the free variables of the function. However, verification and reasoning about functional properties is not a primary goal of our work.

Reachability Analyses. Reachability analyses [Reps 1997; Reps et al. 1995; Vardoulakis and Shivers 2011] have different goals from our work, despite the similarity in names. Those analyses compute whether a program state is reachable from an initial state. In contrast, our reachability type system is concerned with describing potential aliasing, capturing, separation, and other memory properties.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have studied the approach of tracking reachability sets in types to enable ownership-style reasoning for higher-order functional programs. The reachable set of a function is consistent with an interpretation of functions as closure records, and a notion of kill effects suffices to support nested mutable state, unique references, move semantics, linearity, and more. We have formalized the first reachability type-and-effect system and proved a type-and-reachability safety theorem. Our case studies and extensions underline the usefulness and practicality of our system.

In the future, we plan to investigate how to leverage the reachability type-and-effect system to reason about effect dependencies for compiler optimizations based on graph-based intermediate representations [Rompf and Odersky 2012; Rompf et al. 2013].

ACKNOWLEDGEMENTS

We thank James Noble and the anonymous reviewers for their insightful comments. We thank Colin Gordon for discussions on effect systems. We also thank Yushuo Xiao for contributions to prototype implementations. This work was supported in part by NSF awards 1553471, 1564207, 1918483, 1910216, DOE award DE-SC0018050, and NSERC grant CRDPJ 543583-19.

REFERENCES

Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. 2008. Semantics of transactional memory and automatic mutual exclusion. In *POPL*. ACM, 63–74. https://doi.org/10.1145/1328438.1328449

Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L^3: A Linear Language with Locations. *Fundam. Informaticae* 77, 4 (2007), 397–449. http://content.iospress.com/articles/fundamenta-informaticae/fi77-4-06

Alexander Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. 2003. Checking and inferring local non-aliasing. In *PLDI*. ACM, 129–140. https://doi.org/10.1145/781131.781146

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *OOPSLA Companion*. ACM, 1015–1022. https://doi.org/10.1145/1639950.1640073

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. https://doi.org/10.1007/978-3-319-30936-1_14

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

Hendrik Pieter Barendregt. 1985. *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.

Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Math. Struct. Comput. Sci.* 6, 6 (1996), 579–612. https://doi.org/10.1017/S0960129500070109

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. https://doi.org/10.1145/3158093

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting algebraic effects. *Proc. ACM Program. Lang.* 3, POPL (2019), 6:1–6:28. https://doi.org/10.1145/3290319

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.* 4, POPL (2020), 48:1–48:29. https://doi.org/10.1145/3371116

Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, Ondřej Lhoták, and Martin Odersky. 2021. Tracking Captured Variables in Types. (May 2021). arXiv:2105.11896 (v1) [cs.PL] https://arxiv.org/abs/2105.11896

Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. 2002. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*. ACM, 211–230. https://doi.org/10.1145/582419.582440

John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS (Lecture Notes in Computer Science, Vol. 2694)*. Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *ECOOP (Lecture Notes in Computer Science, Vol. 2072)*. Springer, 2–27. https://doi.org/10.1007/3-540-45337-7_2

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020a. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 126:1–126:30. https://doi.org/10.1145/3428194

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020b. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *J. Funct. Program.* 30 (2020), e8. https://doi.org/10.1017/S0956796820000027

Walter Bright. 2019. Ownership and Borrowing in D. https://web.archive.org/web/20210105083139/https://dlang.org/blog/2019/07/15/ownership-and-borrowing-in-d/. Accessed: 2021-01-05.

Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996. Representing Control in the Presence of One-Shot Continuations. In *PLDI*. ACM, 99–107. https://doi.org/10.1145/231379.231395

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1991. An Extension of System F with Subtyping. In *TACS (Lecture Notes in Computer Science, Vol. 526)*. Springer, 750–770. https://doi.org/10.1007/3-540-54415-1_73

Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *ECOOP (LIPIcs, Vol. 56)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.5

Arthur Charguéraud. 2020. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.* 4, ICFP (2020), 116:1–116:34. https://doi.org/10.1145/3408998

Cisco Systems, Inc. 2017. *Chez Scheme Version 9 User's Guide - Chapter 6. Control Structures*. Cisco Systems, Inc. Available at https://cisco.github.io/ChezScheme/csug9.4/control.html, Accessed: 2021-04-10.

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. https://doi.org/10.1007/978-3-642-36946-9_3

Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. 2008. Minimal Ownership for Active Objects. In *APLAS (Lecture Notes in Computer Science, Vol. 5356)*. Springer, 139–154. https://doi.org/10.1007/978-3-540-89330-1_11

David G. Clarke and Sophia Drossopoulou. 2002. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*. ACM, 292–310. https://doi.org/10.1145/582419.582447

David G. 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. https://doi.org/10.1007/3-540-45337-7_4

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

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

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015b. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE! 2015)*. ACM, 1–12. https://doi.org/10.1145/2824815.2824816

Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. 2019. Compiling with continuations, or without? Whatever. *Proc. ACM Program. Lang.* 3, ICFP (2019), 79:1–79:28. https://doi.org/10.1145/3341643

Olivier Danvy and Andrzej Filinski. 1989. A Functional Abstraction of Typed Contexts. *Technical Report, DIKU University of Copenhagen, Denmark* (1989).

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming*. ACM, 151–160. https://doi.org/10.1145/91556.91622

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. https://doi.org/10.1007/978-3-540-74130-5_11

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. https://doi.org/10.1007/978-3-540-85373-2_12

Robert DeLine and Manuel Fähndrich. 2001. Enforcing High-Level Protocols in Low-Level Software. In *PLDI*. ACM, 59–69. https://doi.org/10.1145/378795.378811

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. https://doi.org/10.1145/2049706.2049709

Bruce F. Duba, Robert Harper, and David B. MacQueen. 1991. Typing First-Class Continuations in ML. In *POPL*. ACM Press, 163–173. https://doi.org/10.1145/99583.99608

Matthias Felleisen. 1988. The Theory and Practice of First-Class Prompts. In *POPL*. ACM Press, 180–190. https://doi.org/10.1145/73560.73576

David Gay and Alexander Aiken. 1998. Memory Management with Explicit Regions. In *PLDI*. ACM, 313–323. https://doi.org/10.1145/277650.277748

David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *LISP and Functional Programming*. ACM, 28–38. https://doi.org/10.1145/319838.319848

Jean-Yves Girard. 1971. Une Extension De L'Interpretation De Gödel a L'Analyse, Et Son Application a L'Elimination Des Coupures Dans L'Analyse Et La Theorie Des Types. In *Proceedings of the Second Scandinavian Logic Symposium*, J.E. Fenstad (Ed.). Studies in Logic and the Foundations of Mathematics, Vol. 63. Elsevier, 63–92. https://doi.org/10.1016/S0049-237X(08)70843-7

Colin S. Gordon. 2020a. 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. https://doi.org/10.4230/LIPIcs.ECOOP.2020.10

Colin S. Gordon. 2020b. Lifting Sequential Effects to Control Operators. In *ECOOP (LIPIcs, Vol. 166)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:30. https://doi.org/10.4230/LIPIcs.ECOOP.2020.23

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

Colin S. Gordon, Michael D. Ernst, and Dan Grossman. 2012. Static lock capabilities for deadlock freedom. In *TLDI*. ACM, 67–78. https://doi.org/10.1145/2103786.2103796

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. https://doi.org/10.1145/512529.512563

Philipp Haller and Alexander Loiko. 2016. LaCasa: lightweight affinity and object capabilities in Scala. In *OOPSLA*. ACM, 272–291. https://doi.org/10.1145/2983990.2984042

Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *ECOOP (Lecture Notes in Computer Science, Vol. 6183)*. Springer, 354–378. https://doi.org/10.1007/978-3-642-14107-2_17

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *APLAS (Lecture Notes in Computer Science, Vol. 11275)*. Springer, 415–435. https://doi.org/10.1007/978-3-030-02768-1_22

Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect handlers via generalised continuations. *J. Funct. Program.* 30 (2020), e5. https://doi.org/10.1017/S0956796820000040

Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *FSCD (LIPIcs, Vol. 84)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:19. https://doi.org/10.423

0/LIPIcs.FSCD.2017.18

John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. In *OOPSLA*. ACM, 271–285. https://doi.org/10.1145/117954.117975

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. In *POPL*. ACM, 14–26. https://doi.org/10.1145/360204.375719

Andrej Ivaskovic, Alan Mycroft, and Dominic Orchard. 2020. Data-Flow Analyses as Effects and Graded Monads. In *FSCD (LIPIcs, Vol. 167)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:23. https://doi.org/10.4230/LIPIcs.FSCD.2020.15

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

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

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ifaz Kabir, Yufeng Li, and Ondrej Lhoták. 2020. ιDOT: a DOT calculus with object initialization. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 208:1–208:28. https://doi.org/10.1145/3428276

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ICFP*. ACM, 145–158. https://doi.org/10.1145/2500365.2500590

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

Alexander Kogtenkov, Bertrand Meyer, and Sergey Velder. 2015. Alias calculus, change calculus and frame inference. *Sci. Comput. Program.* 97 (2015), 163–172. https://doi.org/10.1016/j.scico.2013.11.006

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (Lecture Notes in Computer Science, Vol. 10201)*. Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Neelakantan Krishnaswami. 2006. Separation logic for a higher-order typed language. In *Workshop on Semantics, Program Analysis and Computing Environments for Memory Management, SPACE*, Vol. 6. 73–82.

Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. 2009. Design patterns in separation logic. In *TLDI*. ACM, 105–116. https://doi.org/10.1145/1481861.1481874

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *POPL*. ACM, 486–499. https://doi.org/10.1145/3009837.3009872

Fengyun Liu, Ondrej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. 2020. A type-and-effect system for object initialization. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 175:1–175:28. https://doi.org/10.1145/3428243

Yi Lu and John Potter. 2005. A Type System for Reachability and Acyclicity. In *ECOOP (Lecture Notes in Computer Science, Vol. 3586)*. Springer, 479–503. https://doi.org/10.1007/11531142_21

John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *POPL*. ACM Press, 47–57. https://doi.org/10.1145/73560.73564

Nicholas D. Matsakis and Felix S. II Klock. 2014. The Rust language. In *HILT*. ACM, 103–104. https://doi.org/10.1145/2663171.2663188

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. https://doi.org/10.1145/2103656.2103722

Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design (Lecture Notes in Computer Science, Vol. 1710)*. Springer, 114–136. https://doi.org/10.1007/3-540-48092-7_6

James Noble. 2018. Two Decades of Ownership Types. In *SPLASH-I*. ACM.

James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *ECOOP (Lecture Notes in Computer Science, Vol. 1445)*. Springer, 158–185. https://doi.org/10.1007/BFb0054091

Martin Odersky. 1991. How to Make Destructive Updates Less Destructive. In *POPL*. ACM Press, 25–36. https://doi.org/10.1145/99583.99590

Martin Odersky and Tiark Rompf. 2014. Unifying functional and object-oriented programming with Scala. *Commun. ACM* 57, 4 (2014), 76–86. https://doi.org/10.1145/2591013

Peter W. O'Hearn. 2003. On bunched typing. *J. Funct. Program.* 13, 4 (2003), 747–796. https://doi.org/10.1017/S0956796802004495

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. https://doi.org/10.1016/S0304-3975(98)00359-4

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. https://doi.org/10.1007/3-540-44802-0_1

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. https://doi.org/10.1145/2983990.2984009

David J. Pearce. 2011. JPure: A Modular Purity System for Java. In *CC (Lecture Notes in Computer Science, Vol. 6601)*. Springer, 104–123. https://doi.org/10.1007/978-3-642-19861-8_7

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

Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Appl. Categorical Struct.* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *ESOP (Lecture Notes in Computer Science, Vol. 5502)*. Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

Thomas W. Reps. 1997. Program Analysis via Graph Reachability. In *ILPS*. MIT Press, 5–19.

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. ACM Press, 49–61. https://doi.org/10.1145/199448.199462

John C. Reynolds. 1974. Towards a theory of type structure. In *Symposium on Programming (Lecture Notes in Computer Science, Vol. 19)*. Springer, 408–423. https://doi.org/10.1007/3-540-06859-7_148

John C. Reynolds. 1978. Syntactic Control of Interference. In *POPL*. ACM Press, 39–46. https://doi.org/10.1145/512760.512766

John C. Reynolds. 1989. Syntactic Control of Inference, Part 2. In *ICALP (Lecture Notes in Computer Science, Vol. 372)*. Springer, 704–722. https://doi.org/10.1007/BFb0035793

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

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

Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, Christoph Koch, Markus Püschel, and Kunle Olukotun. 2015. Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems. In *SNAPL (LIPIcs, Vol. 32)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 238–261. https://doi.org/10.4230/LIPIcs.SNAPL.2015.238

Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ICFP*. ACM, 317–328. https://doi.org/10.1145/1596550.1596596

Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (2012), 121–130. https://doi.org/10.1145/2184319.2184345

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *POPL*. ACM, 497–510. https://doi.org/10.1145/2429069.2429128

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. https://doi.org/10.1007/978-3-642-45221-5_47

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling effect handlers in capability-passing style. *Proc. ACM Program. Lang.* 4, ICFP (2020), 93:1–93:28. https://doi.org/10.1145/3408975

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI*. ACM, 206–221. https://doi.org/10.1145/3453483.3454039

Frederick Smith, David Walker, and J. Gregory Morrisett. 2000. Alias Types. In *ESOP (Lecture Notes in Computer Science, Vol. 1782)*. Springer, 366–381. https://doi.org/10.1007/3-540-46425-5_24

George Steed and Sophia Drossopoulou. 2016. A principled design of capabilities in Pony. *Master's thesis, Imperial College* (2016).

Alexander J. Summers and Peter Müller. 2011. Freedom before commitment: a lightweight type system for object initialisation. In *OOPSLA*. ACM, 1013–1032. https://doi.org/10.1145/2048066.2048142

The Swift Developer Community. 2019. Ownership Manifesto. https://github.com/apple/swift/blob/main/docs/OwnershipManifesto.md. Accessed: 2021-04-09 (ae2a4cca14).

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

David N. Turner, Philip Wadler, and Christian Mossin. 1995. Once Upon a Type. In *FPCA*. ACM, 1–11. https://doi.org/10.1145/224164.224168

Dimitrios Vardoulakis and Olin Shivers. 2011. CFA2: a Context-Free Approach to Control-Flow Analysis. *Log. Methods Comput. Sci.* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:3)2011

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

David Walker, Karl Crary, and J. Gregory Morrisett. 2000. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst.* 22, 4 (2000), 701–771. https://doi.org/10.1145/363911.363923

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect handlers, evidently. *Proc. ACM Program. Lang.* 4, ICFP (2020), 99:1–99:29. https://doi.org/10.1145/3408981

Hirotoshi Yasuoka and Tachio Terauchi. 2009. Polymorphic Fractional Capabilities. In *SAS (Lecture Notes in Computer Science, Vol. 5673)*. Springer, 36–51. https://doi.org/10.1007/978-3-642-03237-0_5

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.* 3, POPL (2019), 5:1–5:29. https://doi.org/10.1145/3290318

Tian Zhao, Jason Baker, James Hunt, James Noble, and Jan Vitek. 2008. Implicit ownership types for memory management. *Sci. Comput. Program.* 71, 3 (2008), 213–241. https://doi.org/10.1016/j.scico.2008.04.001