Typestate via Revocable Capabilities

SONGLIN JIA, Purdue University, USA
CRAIG LIU, Purdue University, USA
SIYUAN HE, Purdue University, USA
HAOTIAN DENG, Purdue University, USA
YUYAN BAO, Augusta University, USA
TIARK ROMPF, Purdue University, USA

Managing stateful resources safely and expressively is a longstanding challenge in programming languages, especially in the presence of aliasing. For example, scope-based constructs like Java's `synchronized` blocks offer ease of reasoning, but they restrict expressiveness and parallelism. Conversely, imperative, flow-sensitive approaches enable fine-grained control, but they require sophisticated typestate analyses and often burden programmers with explicit state tracking.

In this work, we present a novel approach that unifies the ease of scoped reasoning with the expressiveness of imperative typestate management. Our design extends traditional flow-insensitive capability mechanisms to a flow-sensitive setting. In particular, we decouple capability lifetimes from lexical scopes, allowing functions to receive, revoke, or return capabilities in a flow-sensitive manner, building on existing mechanisms for the safety and ergonomics of scoped capability programming.

We implement our approach as an extension to the Scala 3 compiler, leveraging path-dependent types and implicit resolution to enable concise, statically safe, and expressive typestate programming. Our prototype generically supports a wide range of patterns, including file operations, advanced locking protocols, DOM construction, and session types, showing that expressive and safe typestate management can be achieved with minimal extensions to an existing language with capability support.

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

Additional Key Words and Phrases: typestate, capabilities, reachability types, destructive effects, implicit function types

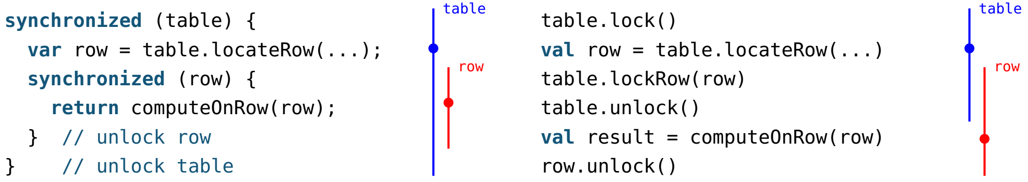
ACM Reference Format:
Songlin Jia, Craig Liu, Siyuan He, Haotian Deng, Yuyan Bao, and Tiark Rompf. 2026. Typestate via Revocable Capabilities. *Proc. ACM Program. Lang.* 10, PLDI, Article 245 (June 2026), 25 pages. https://doi.org/10.1145/3808323

1 Introduction

Programs often perform not only pure computations, but also interact with external environments, observing and mutating *state*. Typical examples include file I/O, remote procedure calls, and thread synchronization. Programming languages support a variety of mechanisms for managing state, balancing ease of reasoning with the expressiveness of complex patterns.

Authors' Contact Information: Songlin Jia, Purdue University, West Lafayette, USA, jia137@purdue.edu; Craig Liu, Purdue University, West Lafayette, USA, liu3477@purdue.edu; Siyuan He, Purdue University, West Lafayette, USA, he662@purdue.edu; Haotian Deng, Purdue University, West Lafayette, USA, deng254@purdue.edu; Yuyan Bao, Augusta University, Augusta, USA, yubao@augusta.edu; Tiark Rompf, Purdue University, West Lafayette, USA, tiark@purdue.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART245
https://doi.org/10.1145/3808323



(a) Written using scope-based **`synchronized`** blocks. Lock lifetimes are managed automatically, whereas `table` has to be locked longer than `row`.

(b) Written in imperative style, `table` can be unlocked once `row` is locked, enabling improved parallelism, at the cost of being explicit about lock lifetimes.

Fig. 1. A database transaction expressed in two different styles, where we first locate a row in the table and then compute a result on the row. Table-level and row-level locks are needed for safe concurrency. In this work, we combine the ease of scoped reasoning (a) with the expressiveness of imperative code (b).

Consider the database transaction illustrated in Figure 1a. It first locates a `row` within a `table` and then computes a result using that `row`. To avoid interference, concurrent access to the `table` and the `row` must be prevented. Languages such as Java provide scope-based constructs like **`synchronized`**, which automatically acquire and release locks upon entering and exiting a scope.

While scoped constructs relieve users from manually managing the state of locks, they lack expressiveness for fine-grained control. As shown in Figure 1b, manual, flow-sensitive management of locks allows the lock for `table` to be released immediately after acquiring the lock for `row`, thereby enabling improved parallelism. In contrast, nested **`synchronized`** blocks impose *last-in-first-out* (LIFO) lifetimes, forcing `table` to be locked longer than `row` and precluding such optimizations.

Nevertheless, neither approach statically enforces that locks are acquired before invoking functions requiring them: programmers may omit **`synchronized`** blocks or lock objects entirely, and such code would remain type-correct yet unsafe in concurrent contexts. In scope-based programming, recent work has introduced mechanisms for providing stronger static guarantees [9, 51, 54, 74, 75]. Among these, *capability-based* designs emerge to bridge ease of use and safety, such that a resource handle carries not only the resource but also a capability to operate on it. Functions like `computeOnRow` can require a lock as a *capability* argument, accessible only within **`synchronized`**:

```
synchronized (row) { lock =>      // given lock
  return computeOnRow(row)(lock)  // using lock (can be inferred in Scala)
}                                 // lock becomes inaccessible
```

Languages such as Scala further advance this paradigm through *implicit argument resolution* [50], which allows capabilities to be supplied automatically when they are in scope.

Establishing static safety guarantees for imperative code, by contrast, necessitates sophisticated *typestate analysis* [65] or *session types* [33]. Without syntactically scoped lifetimes, the type system must precisely track the state of each lock, whether locked or unlocked, at every program point. Methods and functions then need to specify both the required state of their arguments and the state transitions they induce:

```
table.lockRow(row)              // table: Locked required! row: Unlocked to Locked
table.unlock()                  // table: Locked to Unlocked
val result = computeOnRow(row)  // row: Locked required!
row.unlock()                    // row: Locked to Unlocked
```

In addition, conventional type systems are designed to track *invariants* instead of *transitions*. Specialized mechanisms are thus required, and are further complicated by sharing and aliasing.

This Work. We present a solution for flow-sensitive typestate tracking that builds on existing flow-insensitive capability mechanisms, enabling static safety reasoning for expressive imperative

code through minimal extensions. Our approach decouples capability lifetimes from lexical scopes, and allows any function to provide or revoke capabilities in a flow-sensitive manner, thereby supporting effect and state transition tracking independently of any specific typestate discipline.

We have implemented our approach as a prototype by extending the Scala 3 compiler. Capabilities are encoded using path-dependent types [56], enabling precise association between objects and their states. With our extensions, Scala's implicit resolution mechanism facilitates the following three interactions between functions and capabilities:

- **Receiving:** Functions with an *implicit argument arrow* (`?=>`, existing Scala feature [50]) can receive capabilities from the calling context without explicit passing.
- **Revoking:** Functions with a *destructive arrow* (`=!>`) can revoke capabilities (Section 2.3), ensuring they cannot be subsequently accessed (inspired by linear types [71]).
- **Returning:** Functions with an *implicit result arrow* (`?<=`) can return capabilities (Section 2.5), making them available for implicit resolution at the call site of the function (new in this work).

Combining them, a composite arrow (`?=!>?`) expresses state transitions.

Our revocation mechanism relies on a destructive effect system over *descriptive alias tracking*—capturing checking [9, 76, 77] for the implementation, and reachability types [6, 74] for the formal aspects, drawing on the effect system of Deng et al. [18]. Returning capabilities is implemented via a type-directed ANF transformation [57]. Focused on implementation and empirical validation in this work, we leave a complete formal account combining dependent object types [2, 56] with implicit resolution, ANF transformation, and destructive effects as future work.

Our prototype supports a generic typestate system capable of expressing diverse effects and state transitions, including file operations (Section 2), hand-over-hand locks (Figure 1b, Section 3.1), stateful DOM tree construction (Section 3.2), and communication protocols [35, 55] (Section 3.3). Building on the capability-based programming paradigm that accommodates flexible sharing and aliasing, our approach enables concise, readable code without extensive annotations.

The remainder of this paper is structured as follows:

- Section 2 informally introduces our approach step-by-step, through an example showing capability-based programming with files.
- Section 3 presents additional case studies, demonstrating our system on realistic typestate programming scenarios, including locks, DOM trees, and interprocess communication.
- Section 4 discusses implementation details, intended safety properties, and limitations.

Related work is discussed in Section 5, and we conclude in Section 6. We provide additional implementation details in our extended version [36], and our artifact can be found on Zenodo [3].

2 From Capabilities to Typestate, Step by Step

In this section, we informally present our key ideas. We begin by reviewing a scope-based approach to programming with files in Section 2.1. Next, we discuss how to move toward typestate analysis in Section 2.2 and observe its limitations. Finally, we elaborate our three key mechanisms towards safe, expressive, and ergonomic flow-sensitive reasoning in the remainder of this section.

Starting here, we present code in Scala syntax. Our implementation relies on a combination of language features. Scala, as a widely used language, happens to support this particular combination, making it a natural platform for our experiment. Nonetheless, we believe the programming patterns discussed here are language-independent. Mechanisms serving the same goals as implicits appear in other languages in different forms, and alias tracking/controlling mechanisms have also been explored under various contexts. Thus, the principles explored here could inform future designs once similar features become available elsewhere.

2.1 Preliminary: Programming with Scoped Capabilities

Interacting with files and sockets is a common task in daily programming. Many languages provide scoped constructs, such as **with** in Python or **try**-with-resources in Java, to ensure that resources are properly released. In Scala, a typical pattern is to use a higher-order combinator `withFile`:

```
def withFile[T](name: String)(op: File => T): T =  // library code
  val f = File.open(name)
  try op(f) finally f.close()  // grant access to f in op; revoke afterwards
withFile("a.txt"): f =>                               // user code
  f.write("Hello")            // while f is in scope, the file is open
  // f will be closed and go out of scope here
```

A key advantage of using scoped constructs is that the local variable `f` guarantees the underlying file is open for the duration of the scope. Thus, `f` embodies not only the file *resource*, but also the *capability* [19, 46] to operate on it. Unfortunately, this guarantee holds only under conventional, first-order usage. In impure higher-order languages such as Scala, programmers may inadvertently leak these capabilities in various ways:

```
withFile("a.txt") { f => f }                        // Leaked: returned directly
withFile("a.txt") { f => throw f }                  // Leaked: thrown as exception
var f0: File; withFile("a.txt") { f => f0 = f }     // Leaked: stored in mutable vars
withFile("a.txt") { f => () => f.write("Hello") }   // Leaked: captured in closures
```

To prevent such leaks, several mechanisms have been proposed to ensure safe and ergonomic programming using capabilities in Scala. Broadly, they fall into two categories: (1) passing capabilities *implicitly*, so that code never binds them to a named variable; and (2) *explicitly* tracking reachable/captured resources of function closures and data structures at the type level.

2.1.1 Remedy via Implicits. Using implicit function types [50], we can write the same `withFile` example without having the file handle variable in user code. For instance, the scoped file usage can be expressed alternatively:

```
def withFile[T](name: String)(op: File ?=> T) = ... // ?=> for implicit function type
def write(text: String)(using f: File) = ...        // using for implicit arg list
withFile("a.txt") { write("Hello") }                // no explicit binding/passing of f
```

Here `op: File ?=> T` denotes an implicit function type where the `File` argument is supplied implicitly by the compiler. Correspondingly, `write` declares its `File` parameter with a **using** clause, and the file handle variable `f` does not appear in the user code.

The implicit passing style is ergonomic, but it alone has notable limitations. First, it does not scale to multiple files of the same type (like `File`), as the compiler cannot distinguish them, so in practice one still needs explicit names to avoid ambiguity. Second, although the compiler supports implicit resolution, it does not enforce its use, meaning that programmers may still explicitly bind the file handle variable, even when using implicit function types.

```
def mySummon[T](using c: T): T = c     // directly return the implicitly resolved argument
withFile("a.txt") { mySummon[File] }   // Leaked!
```

Since implicits can be bypassed, the file handle `f` can still escape the scope.

As a result, the possibility of capability leakage remains. To obtain static safety guarantees against escaping, additional mechanisms are thus required.

2.1.2 Prevent Leakage by Explicit Typing. Scala has recently introduced an experimental capture checker [48] to prevent unintended escaping of resources. This checker implements a form of *descriptive alias tracking*, in which types are explicitly annotated with variable names to represent the set of resources that may be captured or reached. Such mechanisms have been formally studied

under the names of capturing types (CT) [9, 76, 77] and reachability types (RT) [6, 74]. Although the systems discussed in the literature share foundational concepts, they differ in important aspects such as the treatment of separation and the handling of polymorphism. In this paper, we abstract over these differences by focusing on a common core that suffices for our purposes. We illustrate this shared foundation by demonstrating how it can safely encode the scoped `withFile` pattern.

Qualifiers: Sets of Variables. In both CT and RT, types are accompanied by qualifiers to describe which variables may be captured or reached by a given term. As minimal examples, when new files `fA` and `fB` are opened, their types are annotated with qualifiers containing their respective names:

```
val fA = File.open("a.txt")       // fA: File^{fA}
val fB = File.open("b.txt")       // fB: File^{fB}
```

Qualifiers provide an over-approximation of the variables that a value may capture or reach. For example, consider the variable `fC`, which may alias either `fA` or `fB` depending on a runtime condition. The type of `fC` can be annotated with the qualifier `{fC}`, indicating that it is tracked by its own name, or with `{fA,fB}`, reflecting the possibility that it aliases either `fA` or `fB`. This choice of qualifiers is possible by recording the potential aliasing to both `fA` and `fB` in the typing context.

```
val fC = if (...) fA else fB     // fC: File^{fC}, or File^{fA,fB} ⊣ [..., fC: File^{fA,fB}]
```

Tracking in Higher-Order Functions. In higher-order languages such as Scala, functions may capture free variables or manipulate first-class function closures. Both CT and RT provide mechanisms to reason about such higher-order scenarios. For example, consider an anonymous function that writes to a captured file handle `f`; the outermost qualifier of the function type should include `f`:

```
val f = File.open("a.txt")   // f: File^{f}
() => f.write("Hello")        // <anonymous>: (() => Unit)^{f}
```

More interestingly, consider a function that accepts `f` as a parameter and returns the anonymous function capturing `f`; passing such a function to `withFile` would leak the file handle in a closure. To enable the capture checker for detection, we annotate the type of `f` with `^`, marking it as a tracked resource[1] whose aliasing the compiler will monitor. The separation extension [49] further ensures that `f` carries no undeclared aliases to captured free variables. The result type `(() => Unit)` is annotated with `{f}`, reflecting the escape.

```
def leakFile(f: File^) =      // leakFile: ((f: File^) -> (() ->^{f} Unit)) (CT style)
  () => f.write("Hello")      //           ((f: File♦) => (() => Unit)^{f})^∅  (RT style)
```

When applying `leakFile`, the bound variable `f` in types is substituted with the parameter qualifier:

```
val fA = File.open("a.txt") // fA: File^{fA}
val res = leakFile(fA)      // res: (() => Unit)^{fA} = [f ↦ fA] (() => Unit)^{f}
```

Polymorphism and Leakage Prevention. To specify the type of `withFile`, polymorphism over qualifiers is required. In CT, the type variable `T` is unqualified, and any capturing must be tunneled using *boxes* [12], which are automatically inferred by the compiler. Under this scheme, `withFile` prevents leakage because the boxed `T` cannot carry the tracked file handle out of scope:

```
def withFile[T](name: String)(op: File^ => T): T
withFile("a.txt")(leakFile)  // Error: Ill-scoped unboxing
```

In RT [74], the same guarantee is achieved by explicitly quantifying over both a qualifier `q` and the type `T`, so that the result type `T^q` can only mention variables visible to the caller, excluding the bound file handle:

```
def withFile[T^q](name: String)(op: (File♦ => T^q)♦): T^q
withFile("a.txt")(leakFile)  // Error: Invalid subqual (RT)
```

[1]With different semantics, the freshness maker ♦ in RT-style type signature achieves a similar role.

For the purpose of this paper, we represent type annotations in the RT style to align with the formal model of destructive effects [18], while our code remains compatible with the syntax of Scala capture checking.

2.2 A Naive Step towards Typestate

While scope-based programming offers natural bounds on resource lifetimes, it lacks the expressiveness required for certain scenarios. For example, as illustrated by the lock example in Figure 1, the enforced LIFO discipline can inhibit desirable optimizations and flexible usage patterns.

The same LIFO discipline also restricts flexibility when programming with files. Thus, our objective is to support explicit `open` and `close` operations, eliminating the need for scoped combinators like `withFile`. At the same time, we need the type system to statically guarantee the safe use of file operations with respect to file states. To start with, we define two possible file states as classes:

```
class OpenFile: ...                     // Not directly constructible
class ClosedFile: ...
```

With states defined, we can then define the operations that initialize files and change file states:

```
def newFile(name: String): ClosedFile = ...  // Construct a ClosedFile
def open(f: ClosedFile): OpenFile     = ...  // Transition a ClosedFile to OpenFile
def close(f: OpenFile): ClosedFile    = ...  // Transition an OpenFile to ClosedFile
```

Last but not least, the operations that require files in open states:

```
def read(f: OpenFile): String         = ...
def write(f: OpenFile, text: String): Unit = ...
```

With this API, we can express the same example previously demonstrated with `withFile`, now using explicit typestate transitions. By distinguishing file states with separate types, we statically guarantee that operations like `write` are only permitted for files in appropriate, opened states:

```
val fNew = newFile("a.txt")
val fOpen = open(fNew)
write(fOpen, "Hello")                  // Good: Permitted only after opening the file
val fClosed = close(fOpen)
```

Nevertheless, this initial approach to typestate reasoning exhibits some significant shortcomings:

I: Lack of invalidation for outdated capabilities. After invoking `close(fOpen)`, the variable `fClosed` represents the closed state of the file. However, the original variable `fOpen` remains valid in the type system, allowing subsequent operations such as `write(fOpen, ...)` to type-check, even though the file has already been closed. This permits erroneous use of stale references:

```
val fClosed = close(fOpen)             // fClosed supersedes fOpen
write(fOpen, "Hello")                  // Stale but type-correct
```

II: Inability to verify resource identity. Because each state transition produces a new variable, the type system does not enforce that operations are performed on the intended resource. For example, it is possible to mistakenly operate on the wrong file without detection:

```
val fA = newFile("a.txt"); val fB = newFile("b.txt")
val fOpen = open(fA)                   // a.txt is now open, but not b.txt
write(fOpen, "this should go to b.txt")    // Unintended but type-correct
```

III: Poor ergonomics. This programming style requires explicit threading of stateful objects through sequences of function calls, resulting in cumbersome code.

To overcome these limitations, we propose a flexible and expressive typestate framework grounded in three key mechanisms. First, we introduce a destructive effect system that statically

tracks the revocation of capabilities. Second, we employ path-dependent capabilities to preserve resource identity across state transitions. Third, we leverage an A-normal form (ANF) transformation to enable flow-sensitive implicit resolution. In the rest of this section, we elaborate on these pillars.

2.3 Pillar I: Flow-Sensitive Revocation of Capabilities

To statically invalidate outdated capabilities, we introduce a flow-sensitive *destructive effect* system. This system uses the annotation `@kill(...)` on function result types to specify a set of variables, free ones or arguments, whose use should be prohibited after the function is applied:

```
def newFile(name: String): ClosedFile^        = ...
def open(f: ClosedFile^): OpenFile^ @kill(f)  = ...
def close(f: OpenFile^): ClosedFile^ @kill(f) = ...
```

The effect system sequentially tracks the accumulated set of killed variables, represented as $\{\mathscr{k} : \ldots\}$ below. Invoking the effectful function `close` amounts to extending it with the parameter `fOpen`:

```
val fClosed = close(fOpen)                // {𝓀 : ..., fOpen} extended
write(fOpen, "Hello")                     // Error: found using killed var fOpen
```

Unlike linear type systems [64, 71], where any function using a linear resource must consume it, the `@kill` annotation in our system provides selective, opt-in revocation. Functions that merely use capabilities without revoking them (*e.g.*, `write`) do not induce any destructive effect. This design enables seamless integration with imperative and higher-order constructs:

```
val messages: Array[String] = ...
val fOpen = open(newFile("a.txt"))        // open is effectful
for (msg <- messages) write(fOpen, msg)   // loop of writes is free of kill!
close(fOpen)                              // close is effectful
```

Foundation: Reachability Types and Transitive Disjointness. When capabilities are not required to be linear, sharing and aliasing become possible. In this setting, our effect system must ensure that all potential aliases of a killed capability are also invalidated to provide strong static guarantees. Our model reasons about aliases with reachability types [18]. For example, consider the following scenario, where `fC` may alias either `fA` or `fB`, as recorded in the typing context:

```
val fA = open(newFile("a.txt")); val fB = open(newFile("b.txt"))
val fC = if (...) fA else fB              // context: [..., fC: OpenFile^{fA,fB}]
close(fA)                                 // {𝓀 : ..., fA} extended
write(fC, "maybe to a.txt")               // Error: found using killed var fA
```

After closing `fA`, subsequent writes to `fC` are unsafe, as `fC` may alias the now-closed file. On the other hand, the accumulated set of killed variables is extended by only `fA`, not `fC`. To prevent this misuse, we require that the qualifier of any used term be *transitively disjoint* from the killed set. In this example, the typing context reveals that `fC` may reach both `fA` and `fB`; thus, the separation check fails due to the presence of `fA` in the transitive closure of `fC`'s qualifier:

$$\texttt{fC}* \cap \mathscr{k}* \quad = \quad \{\texttt{fA},\texttt{fB},\texttt{fC}\} \cap \{\ldots,\texttt{fA}\} \quad = \quad \{\texttt{fA}\} \quad \not\subseteq \quad \emptyset$$

To sum up, closing `fA` disables access to both `fA` and any variable that may reach it, such as `fC`, while leaving `fB` unaffected. In contrast, closing `fC` disables access to all three variables: `fA`, `fB`, and `fC`.

An alternative perspective on the effect separation check is provided by continuation-passing style (CPS) [16]. In this formulation, APIs such as `close` and `write` are extended to accept an explicit continuation parameter. Without any effect system, reachability types alone can prevent reusing revoked `OpenFile` handles by enforcing transitive disjointness between the continuation `k` and the handle `f` when both are annotated with the fresh qualifier ♦:

```
// closeCPS: (f: OpenFile^♦) => (k: (ClosedFile^♦ => NoReturn)^♦) => NoReturn
closeCPS(fA){ fA_ => writeCPS(fC, "maybe to a.txt"){...} }   // Error: f,k overlap on {fA}
```

Back in direct-style programming, where there is no explicit notion of continuations, we employ effect tracking and effect separation instead. In CPS, borrowing a resource implicitly becomes permanent because the continuation never returns, which makes a borrow behave effectively like a move. Our direct-style kill effects mirror this observation by ensuring that once a capability is revoked, all reachable aliases are invalidated.

Notation. We represent the types of functions that kill their arguments using the arrow `=!>` and its implicit variant `?=!>`. The signatures of `open` and `close` can be simplified accordingly.

```
type =!>[S, T]  = (c: S) =>  T @kill(c)      // arrow type     that kills arg c
type ?=!>[S, T] = (c: S) ?=> T @kill(c)      // implicit arrow that kills arg c
// open: ClosedFile♦ =!> OpenFile♦; close: OpenFile♦ =!> ClosedFile♦
```

2.4 Pillar II: Relating Capabilities and Objects by Path-Dependent Types

While the effect system ensures that new states of an object supersede previous ones, it does not distinguish between states of different objects. To illustrate this limitation, consider a variant of the `withFile` combinator, named `ensureClosed`, which operates over the state class `ClosedFile`:

```
def ensureClosed(name: String)(op: ClosedFile^ =!> ClosedFile^): Unit =
  op(newFile(name)); ()
```

Here, the type of `op` enforces that a fresh `ClosedFile` is returned, thereby requiring that any file opened within the scope of `ensureClosed` must be closed before the function returns:

```
ensureClosed("a.txt"): f =>
  val fOpen = open(f)
  write(fOpen, "Hello")
  close(fOpen)           // Error if omitted
```

However, the type system does not guarantee that the `ClosedFile` returned by `op` is the same file that was originally provided. For example, the type checker would accept an instance of `op` that simply returns a newly created, unopened file, rather than the intended one:

```
ensureClosed("a.txt"): f =>
  val fOpen = open(f)
  newFile("b.txt")       // Unintended but type-correct
```

More fundamentally, even if `newFile` is removed from the API, a programmer can still circumvent the intended guarantees by reusing a `ClosedFile` obtained from an outer `ensureClosed` block. A rigorous mechanism relating capabilities and object identities is necessary.

Why not Reachability/Capturing Types. Although descriptive alias tracking mechanisms offer promising ways to reason about resources, they do not address this identity problem. As illustrated by the signatures of `open` and `close` above, these APIs always revoke the provided capability and generate a fresh one; at the type level, no relationship is maintained between the input and output capabilities. Attempting to relate them would result in the returned capability being immediately invalidated by the kill effect. Furthermore, both RT and CT are inherently over-approximations: a qualifier `{f}` indicates that a capability may, but does not necessarily, refer to `f`. Consequently, these systems cannot provide the desired safety properties (Section 4.1) here.

Our Solution: Path-Dependent Capabilities. Independent of reachability types and effects that govern the lifetime of capabilities, we leverage *path-dependent types* from Dependent Object Types (DOT) [2, 56] to track capability identities. In Scala, a class may declare *abstract type members*: types that are left unspecified in the class definition and whose concrete identity is determined per object instance. Because each instance carries its own copy of these types, they are effectively existential: for two variables `a` and `b` of the same class, the *path-dependent* types `a.T` and `b.T` are

considered distinct by the compiler. We exploit this property to represent files as a unified class `File`, with the two possible states as abstract type members:

```
class File:
  type IsClosed          // abstract type members
  type IsOpen            // for f of type File, there are caps: f.IsClosed and f.IsOpen
```

Crucially, for any two distinct variables `f` and `g` of type `File`, the corresponding path-dependent types `f.IsClosed` and `g.IsClosed` are also distinct and cannot be confused by the type system. This property enables us to define file APIs in a path-dependent manner:

```
def openDep(f: File, c: f.IsClosed^): f.IsOpen^ @kill(c)   = ...
def closeDep(f: File, c: f.IsOpen^):  f.IsClosed^ @kill(c) = ...
def readDep(f: File, c: f.IsOpen^):   String               = ...
def writeDep(f: File, s: String, c: f.IsOpen^): Unit       = ...
```

In these APIs, `f` denotes the file *resource*, while `c` is a *path-dependent capability* whose type is prefixed by the specific variable `f`. The transition functions `openDep` and `closeDep` consume (kill) their input capabilities and return fresh ones, all associated with the same file via the path prefix `f`.

The scoped combinator `ensureClosedDep` additionally provides the initial capability and enforces that the returned capability corresponds to the same file, thereby guaranteeing that resources are properly closed upon exiting the scope and preventing confusion between object identities:

```
def ensureClosedDep(name: String)(op: (f: File) => f.IsClosed^ =!> f.IsClosed^): Unit =
  ...
ensureClosedDep("a.txt"): f => cInit =>
  val cOpen = openDep(f, cInit)
  writeDep(f, "Hello", cOpen)
  closeDep(f, cOpen)            // Error if omitted
ensureClosedDep("a.txt"): f1 => cInit1 =>
  val cOpen1 = openDep(f1, cInit1)
  ensureClosedDep("b.txt"): f2 => cInit2 =>
    closeDep(f1, cOpen1)        // Error: expect f2.IsClosed, got f1.IsClosed
```

Crucially, safe use of such APIs requires disciplined encapsulation. To keep the type members of `File` abstract, it is critical that we only introduce files via `ensureClosedDep`, whereas common constructors could instantiate capabilities with arbitrary types. This can be enforced with appropriate encapsulation techniques (e.g. **`private`** constructors) orthogonal to our presentation.

Bundling Resources and Capabilities as Σ. A final challenge remains in adapting the `newFile` operation to the path-dependent capability style. Unlike `ensureClosedDep`, which supplies both the file and its initial `IsClosed` capability as separate, yet dependent, arguments within a scope, the imperative `newFile` must return both the file object and its associated capability together, while preserving their type-level dependency. This necessitates a mechanism for simultaneously constructing and returning a resource and its path-dependent capability in a type-safe manner.

A natural solution to this problem is dependent pairs, also known as Σ types. To achieve this, we define a **`trait`**, Scala's analogue of an interface, named `Sigma`:

```
trait Sigma { type A; type B; val a: A; val b: B }
              \_____________/ \_______________/
              abstract types  fields typed by A, B
```

To use it as the result type of `newFileSigma`, `Sigma` can be refined with concrete, dependent `A` and `B`. We instantiate `A` with the type of resources, `File` here. Crucially, we instantiate `B` as the type of the path-dependent capabilities by referring to the value field `a` within `Sigma`:

```
def newFileSigma(name: String): Sigma { type A = File; type B = a.IsClosed^ } = ...
```

Crucially, Sigma should be understood as a *transient* wrapper for bundling resources and capabilities, backed by specialized compiler support, but not a dependent type data structure with reachability tracking, which is beyond the scope of this work. Once returned from newFileSigma, the result sigma needs immediate unpacking to maintain sound reachability tracking. To preserve the type-level dependency, we need to ascribe the field, a, using singleton types [53]:

```
val sigma = newFileSigma("a.txt")
val f: sigma.a.type = sigma.a    // sigma.a.type: singleton type of sigma.a
val c = sigma.b                  // c: f.IsClosed^{c}
```

2.5 Pillar III: Flow-Sensitive Introduction of Capabilities

While the combination of destructive effects and path-dependent capabilities yields strong safety guarantees, programming directly with these mechanisms can be verbose and unwieldy. In this section, we present a series of techniques to improve the ergonomics of typestate programming, enabling more concise and user-friendly code without compromising safety.

Implicit Resolution. To enable implicit argument resolution [50] for our APIs, we can declare the capability argument in a separate argument list led by the **using** keyword:

```
def openImp(f: File)(using c: f.IsClosed^): f.IsOpen^ @kill(c) = ...
```

Or, more concisely, using the notations for implicit arrows and destructive arrows:

```
def openImp(f: File):  f.IsClosed^ ?=!> f.IsOpen^    = ...  // ?=!>: implicit + kill
def closeImp(f: File): f.IsOpen^   ?=!> f.IsClosed^  = ...
def readImp(f: File):  f.IsOpen^   ?=>  String       = ...  // ?=>: implicit only
def writeImp(f: File, s: String): f.IsOpen^ ?=> Unit = ...
```

With these APIs leveraging implicit resolution, capabilities no longer require explicit passing. However, implicit instances must still be declared explicitly, which introduces additional complexity. In particular, unpacking the bundled Sigma type requires singleton type ascription, and careful scoping is needed to disambiguate multiple live capabilities of the same type, including revoked ones. Without additional mechanisms, programmers must manually structure scopes to maintain unambiguous implicit resolution, which is undesirable.

Σ-Guided ANF Transformation. To facilitate the ergonomic, flow-sensitive introduction of path-dependent capabilities encapsulated within Sigma types, we employ a type-directed A-normal form (ANF) transformation [57]. Specifically, for any non-tail expression of type Sigma, the transformation restructures the continuing computation into a new block. Within this block, the first field a is extracted and ascribed a singleton type, while the second field b is declared as an implicit candidate. This block-based approach ensures that the newly introduced implicit has the highest precedence in subsequent resolution, thereby eliminating ambiguity and supporting reliable capability inference:

```
val f = newFileSigma()
openImp(f)
```

⟹

```
val sigma_0 = newFileSigma()
{
  implicit val sigma_0_imp = sigma_0.b
  val f: sigma_0.a.type = sigma_0.a
  openImp(f)   // inferred using sigma_0_imp
}
```

More generally, other APIs can be refactored to return a Sigma type, enabling their use to benefit from the ANF transformation described above. For example, a variant of open may return the new capability as the second field of a Sigma, with the first field instantiated as Unit:

```
def openSigma(f: File): f.IsClosed^ ?=!> Sigma { type A = Unit; type B = f.IsOpen^ } = ...
```

Implicit Σ-Lifting. To further streamline the construction of `Sigma` results, we introduce implicit Σ-lifting. This mechanism is particularly beneficial for combinators such as `ensureClosedSigma`, which require the callback `op` to return both a data value of type `T` and an `IsClosed` capability witness:

```
def ensureClosedSigma[T](name: String)
  (op: (f: File) => f.IsClosed^ ?=!> Sigma { type A = T; type B = f.IsClosed^ }): T = ...
```

When returning the result read from the file, it is natural for users to simply return the string variable `text`. However, the expected return type is a dependent pair (`Sigma`), requiring the result and a capability to be bundled together. To reconcile this mismatch, the compiler automatically lifts the return value into the first field `a` of the `Sigma` pair, while the second field `b` is populated by implicitly summoning the appropriate capability. This implicit Σ-lifting mechanism ensures that the returned value conforms to the required dependent pair type without additional user intervention:[2]

```
ensureClosedSigma("a.txt"): f => c ?=>
  openSigma(f)(using c)
  val text = readSigma(f)
  closeSigma(f)
  text  // needs lifting
```

⟹

```
ensureClosedSigma("a.txt"): f => c ?=>
  ...
  new Sigma:
    type A = String
    type B = f.IsClosed^
    val a = text
    val b = summon[f.IsClosed]  //<-closeSigma
```

Given the ANF transformation creating a new scope for `closeSigma`, the summon can locate the most recent, live capability for `f.IsClosed`.

Notation. As a dual to implicit function types (`?=>`), which receive implicit arguments, the Σ-guided ANF transformation enables the implicit return of results, thus complementing the flow-sensitive revocation of capabilities in Section 2.3. To make this duality explicit, we introduce the arrow notation `?<=` as an alternative to `Sigma`:

```
type ?<=[B1, A1] = Sigma { type A = A1; type B = B1 }
// openSigma: (f: File) => f.IsClosed♦ ?=!> f.IsClosed♦ ?<= Unit
```

Going further, for functions such as `openSigma` that perform only state transitions and do not produce an explicit output, we introduce the combined arrow notation `?=!>?`. This notation succinctly expresses three key aspects: (1) the function receives an implicit capability argument, (2) the capability is revoked via a destructive effect, and (3) a new implicit capability is returned. This abstraction streamlines the specification of typestate transitions, improving both the clarity and conciseness of API signatures.

```
type ?=!>?[S1, S2] = (c: S1^) ?=!> ((S2^) ?<= Unit)
// openSigma:  (f: File) => f.IsClosed ?=!>? f.IsOpen
// closeSigma: (f: File) => f.IsOpen   ?=!>? f.IsClosed
```

2.6 Summary

In this section, we have developed a generic typestate framework by progressively extending scoped capability-based file programming with three key mechanisms:

- **Flow-sensitive revocation** (Section 2.3): a destructive effect system that statically invalidates outdated capabilities, ensuring that stale references cannot be used after a state transition.
- **Path-dependent capabilities** (Section 2.4): abstract type members that relate each capability to a specific object instance, preventing confusion between the states of different resources.
- **Σ-guided ANF transformation** (Section 2.5): a type-directed rewriting that automatically introduces and resolves capabilities via implicits, eliminating manual threading.

[2]Due to current limitations in Scala regarding curried dependent implicit function types, the implicit parameter `c` must be explicitly bound and passed. This restriction is incidental to our approach; for clarity, we omit them in subsequent examples.

To illustrate how these pieces fit together, the file API developed throughout this section is summarized below, alongside a usage example in its final, ergonomic form:

```
// API
def newFile(name: String): Sigma { type A = File; type B = a.IsClosed^ }
def open(f: File):              f.IsClosed ?=!>? f.IsOpen    // receive, revoke, return
def close(f: File):             f.IsOpen   ?=!>? f.IsClosed
def read(f: File):              f.IsOpen   ?=>   String
def write(f: File, s: String): f.IsOpen   ?=>   Unit
// Usage
val f = newFile("a.txt")        // f: File, f.IsClosed introduced
open(f)                         // f.IsClosed consumed, f.IsOpen introduced
write(f, "Hello")               // f.IsOpen summoned
close(f)                        // f.IsOpen consumed, f.IsClosed introduced
```

In the rest of the paper, we present additional case studies to illustrate the use of our framework across diverse domains and elaborate on the underlying design and implementation.

3 Case Studies

In this section, we present additional case studies to demonstrate our programming model. To keep the presentation focused on key concepts, we defer low-level implementation details to our extended version [36]. All definitions and examples presented in this section are fully supported and type checked in our prototype implementation [3].

3.1 Table Locking

Our first case study revisits the imperative table locking example from Figure 1b in Section 1. Figure 2 presents one possible implementation, including the `Table` definitions and the corresponding API. To track the lock status of tables and rows, we define a mixin `Lock` (line 1) containing a type member for each state. Both the `Table` (line 5) and its nested class `Row` (line 8) extend `Lock`, so that

```
1  trait Lock:
2    type IsHeld      // lock held,     resource usable
3    type IsReleased  // lock released, resource unusable
4
5  class Table private[...](n: Int) extends Lock:  // constructor private to API package
6    private val data = ...   // an indexable data storage
7    // ... table lock fields ...
8    class Row private[...](m: Int) extends Lock:  // constructor private to API package
9      private val row = data(m)  // mth row of table
10     // ... row lock fields ...
11
12 object Table:
13   def apply(n: Int): Sigma { type A = Table; type B = a.IsReleased^ } = ...  // factory method
14
15 extension (table: Table)
16   def lock():   table.IsReleased ?=!>? table.IsHeld = ...  // acquire the lock of table
17   def unlock(): table.IsHeld ?=!>? table.IsReleased = ...  // release the lock
18   def locateRow(n: Int): table.IsHeld^ ?=> Sigma { type A = table.Row; type B = a.IsReleased^ } = ...
19     // locate nth row of table
20   def lockRow(row: table.Row): table.isHeld^ ?=> row.IsReleased ?=!>? row.IsHeld = ...
21     // acquire the lock of row
22
23 extension (row: Table#Row)
24   def unlock(): row.IsHeld ?=!>? row.IsReleased = ...        // release row lock
25 def computeOnRow(row: Table#Row): row.IsHeld^ ?=> ... = ...  // compute on row
```

Fig. 2. Table Locking Definitions and API

each table and row carry their own lock states. Because `Row` is defined inside `Table`, every `Row` instance is statically tied to its parenting `Table`.

The factory method on line 13 creates a new `Table` together with its initial capability `isReleased`, packaged as a `Sigma` pair. The `Table` itself is returned explicitly, while the `isReleased` capability is returned implicitly. Line 15 defines a collective extension for `Table` instances, allowing additional methods to be invoked in the standard object-oriented style, such as `table.lock()`.

Methods that perform typestate transitions use the composite arrow `?=!>?`. For example, `lock` (line 16) acquires the lock on a `Table` by consuming the `table.IsReleased` capability and returning a `table.IsHeld` capability. Then, operations that need `Table` to be in a specific state are implicitly parameterized by the corresponding path-dependent capability. Retrieving a row (line 19) requires the `Table` lock to be held, so it requires the `table.IsHeld` capability, and returns a `Row` value which depends on the same `Table` instance, together with its `isReleased` capability.

Notably, to lock a `Row`, the enclosing `Table` must already be locked, but unlocking a `Row` does not require so. Accordingly, `lockRow` (line 20) operates on a specific `table` instance and requires the `table.isHeld^` capability, whereas `unlock` (line 24) requires no specific table. This independence is expressed using the *type projection* `Table#Row` [53], which is used to refer to arbitrary `Row` objects.

3.2 DOM Trees

The examples so far have been modeled as finite-state machines, while our programming model is also capable of tracking typestate defined by a context-free grammar. To demonstrate this expressiveness, we sketch a stateful API for constructing DOM trees, where the typestate corresponds to a list of currently open brackets:[3]

```
makeDOM: dom =>
  dom.open(DIV())
  // ... adding text to DIV ...
  dom.close(P())     // Error: state is [DIV] not [P, ...]
  dom.close(DIV())
  dom.close(DIV())  // Error: state is [], not [DIV, ...]
```

Figure 3 depicts a minimal set of interfaces required to track such typestate. Tracking a list of open brackets necessitates expressing lists of DOM elements as types. We first introduce a sum type `Elem` (line 1), with one variant per DOM node. Subsequently, we define a *heterogeneous type list* [39] `EList` (line 2) of `Elem` types. The `DOM` class (line 5) possesses a higher-kinded type member `Elems` parameterized by a `EList`. Different `DOM` states are hence different `EList` parameters to `Elems`.

```
1  trait Elem; class DIV() extends Elem; class P() extends Elem; ... // other elements
2  trait EList; class ENil extends EList
3  class ::[E <: Elem, L <: EList] extends EList                    // :: can be used as infix
4
5  class DOM private[...]() { type Elems[L <: EList];  ... /* other fields */ }
6
7  extension (dom: DOM)
8    def open[E <: Elem, L <: EList](elem: E):  dom.Elems[L] ?=!>? dom.Elems[E :: L] = ...
9    def close[E <: Elem, L <: EList](elem: E): dom.Elems[E :: L] ?=!>? dom.Elems[L] = ...
10   def text[E <: Elem, L <: EList](elem: E, s: String): dom.Elems[E :: L]^ ?=> Unit = ...
11
12 def makeDOM(body: (dom: DOM) => (dom.Elems[ENil]^) =!> (dom.Elems[ENil]^) ?<= Unit): Unit = ...
```

Fig. 3. DOM Definitions and API

[3]Due to Scala limitations with curried implicit dependent function types, the first line should bind the capability explicitly, *i.e.*, `dom => c =>`. As before, we omit `c` as it is incidental to our approach.

```
makeDOM: dom =>
  dom.open(DIV())
  dom.close(DIV())
  dom.close(DIV())  // Error
```

(a) Error on line 4, due to use of killed variable with type `dom.Elems[DIV :: ENil]`

```
makeDOM: dom =>
  dom.open(DIV())
  dom.close(DIV())
  dom.close(HEAD())  // Error
```

(b) Error on line 4, since no implicit found of type `dom.Elems[HEAD :: ...]`

Fig. 4. Errors caught by the DOM API

```
1   // creates </tr><tr>
2   def nextTR[L <: EList](t: DOM): (t.Elems[TR :: L] ?=!>? t.Elems[TR :: L]) =
3     t.close(TR()); t.open(TR())
4
5   // creates <td>p._1</td> <td>p._2</td>
6   def twoCells[L <: EList](t: DOM, f: String, s: String): t.Elems[TR :: L] ?=!>? t.Elems[TR :: L] =
7     t.open(TD()); t.text(TD(), f); t.close(TD());
8     t.open(TD()); t.text(TD(), s); t.close(TD())
9
10  def main() = makeDOM { dom =>
11    dom.open(TABLE()); dom.open(TBODY()); dom.open(TR())
12    val response = await(fetch("/api/logs").toFuture) // fetch log file
13    val reader = response.body.getReader()            // get log file reader
14    type TStart = TR :: TBODY :: TABLE :: ENil
15
16    def readAll(dom: DOM): dom.Elems[TStart] ?=!>? dom.Elems[ENil] =
17      val chunk = await(reader.read().toFuture)       // read chunk from log file
18      if (!chunk.isDone) then
19        val line = chunk.value                        // get line from chunk
20        twoCells(dom, line.timeStamp, line.message); nextTR(dom)
21        readAll(dom)
22      else
23        dom.close(TR()); dom.close(TBODY()); dom.close(TABLE()) // close table
24    readAll(dom) }
```

Fig. 5. Building a HTML table containing log file lines using the DOM API.

The `DOM` object tracks two kinds of state changes: opening and closing elements. Opening an element `E` transitions the `EList` state from `L` to `E :: L`, prepending `E` to the list. Conversely, closing an element (line 9) performs the dual operation, dropping `E`. This mechanism statically enforces correct bracketing, that is, each opened element must be closed in the reverse order.

To introduce `DOM` objects, the API provides a higher-order function `makeDOM` (line 12), analogous to `withFile` to ensure that the `DOM` tree is fully bracketed. Its `body` parameter takes a `DOM` object `dom` and a `dom.Elems[ENil]` capability, and must return the same capability using `Sigma`, guaranteeing that all opened elements are eventually closed. This design detects several kinds of errors at compile-time, such as double element closure (Figure 4a) or closing an element (Figure 4b) that is never opened.

DOM API Example. Figure 5 demonstrates how this API enables precise control over opening and closing `DOM` trees. The example builds an HTML table where each row is a separate line of a log file, assuming a standard HTTP request API capable of fetching data asynchronously.

Before `main`, auxiliary methods `nextTR` (line 2) and `twoCells` (line 6) are defined. Using manual open and close operations, `nextTR` creates a non-bracketed row transition `</tr><tr>`, and `twoCells` constructs bracketed columns. The two functions both use `?=!>?` to indicate that they perform state transitions on the argument.

The `main` function first opens the HTML table elements before fetching the log file. It then defines the recursive method `readAll` (line 16), which repeatedly reads from the log file and creates a new table row for each line of the log file. The `readAll` method takes a `DOM` object and transitions its

```
1  trait Local  // Extended by types below
2  class Send[B, T <: Local] ...
3  class Recv[B, T <: Local] ...
4  class Branch[L <: Local, R <: Local] ...
5  class Select[L <: Local, R <: Local] ...
6  class End ...
7  class Rec[T <: Local] ...
8  class Var[N <: Int] ...
9
10 trait PList
11 class PNil extends PList
12 class ::[T <: Local, L <: PList] extends PList
13 class Chan private[...](): // Channel
14   type PCap[E <: PList, T <: Local]
15   // ... more fields
16
17 type Dual[T <: Local] <: Local = T match
18   case Send[b, t] => Recv[b, Dual[t]]
19   case Recv[b, t] => Send[b, Dual[t]]
20   case Branch[l, r] => Select[Dual[l], Dual[r]]
21   case Select[l, r] => Branch[Dual[l], Dual[r]]
22   case End => End
23   case Rec[t] => Rec[Dual[t]]
24   case Var[n] => Var[n]
25 object Chan:
26   def apply[T <: Local](): // factory method, creates two dual channels
27     ( Sigma { type A = Chan; type B = a.PCap[PNil, T]^ },
28       Sigma { type A = Chan; type B = a.PCap[PNil, Dual[T]]^ }) = ...
29
30 // type parameter bounds omitted: E <: PList, and T, L, R all <: Local
31 extension (chan: Chan)
32   // basic Channel Operations
33   def send[B, E, T](x: B): chan.PCap[E, Send[B, T]] ?=!>? chan.PCap[E, T] = ... // sends x: B
34   def recv[B, E, T](): (chan.PCap[E, Recv[B, T]]^) ?=!> ((chan.PCap[E, T]^) ?<= B) = ... // recv B
35   def close[E](): chan.PCap[E, End] ?=!> Unit = ...
36   // channel choice
37   def left[E, L, R]():  chan.PCap[E, Select[L, R]] ?=!>? chan.PCap[E, L] = ... // choose left
38   def right[E, L, R](): chan.PCap[E, Select[L, R]] ?=!>? chan.PCap[E, R] = ... // choose right
39   def branch[E, L, R, F](using c: chan.PCap[E, Branch[L, R]]^)
40       (l:  chan.PCap[E, L] ?=!> F)(r: chan.PCap[E, R] ?=!> F): F @kill(c) =
41     if (...) then l(...) else r(...)  // invokes l or r depending on received choice
42   // protocol recursion
43   def recPush[E, T](): chan.PCap[E, Rec[T]] ?=!>? chan.PCap[T :: E, T] = ...
44   def recTop[E, T](): chan.PCap[T :: E, Var[0]] ?=!>? chan.PCap[T :: E, T] = ...
45   def recPop[E, T, N <: Int](): chan.PCap[T :: E, Var[S[N]]] ?=!>? chan.PCap[E, Var[N]] = ...
```

Fig. 6. Session Type and Chan API Definitions

typestate from `TStart`, the initial state of the table, to `ENil`, representing a fully closed tree. Each iteration reads a new chunk via the asynchronous reader. If the chunk is not yet complete (`isDone`), it extracts the log line and uses `twoCells` to emit two table cells (timestamp and message), and then calls `nextTR` to start a new row. The recursive call then proceeds with the updated typestate produced by `nextTR`, which type-checks. In the else branch, `readAll` closes the table and produces the final typestate `dom.Elems[ENil]`. At the end of the `makeDOM` block, `readAll` is invoked, ensuring that all elements are properly closed and the resulting DOM tree is well-bracketed.

3.3 Session Types

Session types [31, 32, 67] are a type-based discipline for specifying and verifying communication protocols, ensuring type safety and adherence to protocols. In particular, *binary session types* regulate communication between two parties, each associated with a type describing its protocol:

```
// protocol of this channel endpoint: Send[String, Send[String, Recv[Int, End]]]
chan.send("Hello"); chan.send("World"); println(chan.recv() + 20)
```

To ensure communication safety, session-typed channel endpoints adhere to the property of *duality* that every message sent by one endpoint is received by the other. The dual of the protocol in the above example is `Recv[String, Recv[String, Send[Int, End]]]`, which the other endpoint of the channel must be typed upon.

We present an implementation of binary session types in our programming model, following Jespersen et al. [35], Pucella and Tov [55], translated into Scala. Our definitions (Figure 6) closely

mirror the standard formulation: the `Local` (line 1) session type and extensions describe the protocol, and the channel class `Chan` (line 13) embeds the protocol `T` into a type member `PCap`, as capabilities. Protocol recursion is represented by an additional *protocol environment* parameter `E` (line 14), which stores protocols later retrievable by de-Bruijn indices. The type `Dual`, recursively computing the duality of a given `Local` type, is implemented using Scala's *match types* [8].

Figure 6 then defines the `Chan` API. The factory method for channels (line 26) returns a pair of `Sigma` objects, each containing a `Chan` object in conjunction with its `PCap` capability. The two endpoints have dual protocols, and both `PCap` capabilities become available as implicits in the caller's scope. The `send` method (line 33) requires a `Send[B, T]`-typed capability, and transitions the protocol to only `T`. Because `recv` (line 34) must also return a value of type `B`, we cannot use `?=!>?` directly. Instead, `?=!>` and `?<=` are used to signify that it takes in a `Recv[B, T]`-typed capability, revokes it, and then returns the remaining session capability implicitly and the received value explicitly.

To handle branching, we must return either an `L`-typed capability or `R`-typed capability. Therefore, `branch` takes two callbacks and invokes one depending on the received choice. To work with Scala's type inference, `branch` must specify an explicit **`using`** clause and a kill effect.

The methods `recPush` (line 43), `recTop` (line 44), and `recPop` (line 45) carry out protocol recursion. First, `recPush` operates on a `Rec[T]`-typed `Chan` by pushing `T` onto the protocol environment `E`. Then, a `Var[K]`-typed `Chan` refers to the protocol `K` deep in `E`. If `K` is `0`, then `recTop` replaces `Var[0]` with the top of `E`. Otherwise, `K` could be the successor of a natural number `N`, expressed in Scala as `Var[S[N]]`. Here, `recPop` can be used to remove the top of `E` and decrement `N`.

Echo Server/Client. We demonstrate this API in Figure 7. The server will receive a string, print it and then, depending on client choice, either repeat or quit. This protocol can be defined as:

```
type EchoSInner = Recv[String, Branch[Var[0], End]]
type EchoServer = Rec[EchoSInner]; type EchoClient = Dual[EchoServer]
```

The methods `echoServer` and `echoClient` require a `PCap` capability for their respective protocols, and to perform state transitions on the channel, they must also kill it, resulting in a full signature of:

```
def echoServer(chan: Chan): chan.PCap[PNil, EchoServer] ?=!> Unit
```

The inner `recur` methods also have a similar return type signature. Then, we can create a channel and run `echoServer` and `echoClient` in parallel (Figure 7c). The method `cFuture` is a wrapper for creating

```
def echoServer(chan: Chan): ... =
  chan.recPush()
  def recur(chan: Chan): ... =
    println(chan.recv())
    chan.branch {
      chan.recTop(); recur(chan)
    } {
      chan.close()
    }
  recur(chan)
```

(a) Echo Server implementation. It prints out the received message and then either repeats or closes depending on client choice.

```
def echoClient(chan: Chan): ... =
  chan.recPush()
  def recur(chan: Chan): ... =
    chan.send(readLine())
    if (readLine() == ...) then
      chan.left(); chan.recTop()
      recur(chan)
    else
      chan.right(); chan.close()
  recur(chan)
```

(b) Echo Client implementation. It reads and sends a message before repeating or closing depending on client choice

```
def main() =
  val (serverChan, clientChan) = Chan[EchoServer]()
  cFuture { echoServer(serverChan) }  // kills serverChan.PCap, a free variable of body
  cFuture { echoClient(clientChan) }  // kills clientChan.PCap
```

(c) Running channels in parallel. Using `cFuture` permits typestate transitions.

Fig. 7. Echo program implementation. The method return types are omitted.

```
1  def ifDiff[T, B1, B2](using c: T^)(cond: => Boolean)[U]
2    (tbranch: (T^) ?=!> ((B1^) ?<= U))
3    (ebranch: (T^) ?=!> ((B2^) ?<= U)): ((Either[B1, B2]^) ?<= U) @kill(c) = ...
4  def matchSame[B1, B2, T](using c: Either[B1, B2]^)[U]
5    (left:  (B1^) ?=!> ((T^) ?<= U))
6    (right: (B2^) ?=!> ((T^) ?<= U)): (T^ ?<= U) @kill(c) = ...
7  def loop[T](using c: T^)(cond: => Boolean)(body: T ?=!>? T):  ((T^) ?<= Unit) @kill(c) = ...
8  def whileLeft[T, U](using c: T^)(body: T ?=!>? Either[T, U]): ((U^) ?<= Unit) @kill(c) = ...
```

Fig. 8. Control Flow Combinators

`Future`. As callbacks passed to `cFuture` may kill a free variable (the captured `PCap` capabilities), `cFuture` possesses an observable kill effect that must be annotated on the callback type. We introduce a *function self-reference* `FUN` for `@kill()`; a function with `@kill(FUN)` will induce a destructive effect on itself, thus allowing it to kill arbitrary free variables. The method `cFuture` is then defined as:

```
def cFuture[T](body: => T @kill(FUN)): Future[T]  // body can only be used once
```

where the `FUN` marker refers to `body` itself. Note that `body` is a by-name parameter: any expression provided to `cFuture` as parameter will be lifted into a parameterless function, subject to destruction by `@kill(FUN)`. Used in Figure 7c, this ensures that the server and the client are both one-shot.

3.4 Control Flow

An important facet of typestate is its interaction with control flow. While our model supports basic typestate use within conditionals (see Figures 5 and 7), some usages are not directly expressible. We address this by defining a series of generic combinators for control flow (Figure 8).

The method `ifDiff` (line 1) is parameterized by a capability of type `T`, meant to represent some arbitrary initial typestate. It then evaluates either `tbranch` or `ebranch` depending on the result of `cond`. Both thunks can transition the initial typestate `T` to some different states, `B1` for `tbranch` and `B2` for `ebranch`. Then, the union of the two typestates is represented by implicitly returning an `Either[B1,B2]`. The method `matchSame` (line 4) is the dual of `ifDiff`. It performs the inverse operation of deconstructing an `Either`, requiring both cases to transition to the same typestate.

Furthermore, `ifDiff` and `matchSame` can be used to define methods for iteration. The method `loop` (line 7) takes in some state `T`, and then repeats `body` while `cond` remains true. Importantly, `body` must maintain the state `T` to ensure sound repetition. The method `whileLeft` (line 8) has no condition; it repeats `body` until the initial state `T` transitions to some other state `U`. The implementation of these combinators [3] illustrates an advantage of using capabilities: since they are merely program values, they can be easily inspected and manipulated by ordinary constructs, *e.g.*, pattern matches.

```
if (...) { f.open() }  // f: File
   else  { f.open(); f.close() }
```

```
ifDiff[f.IsClosed, f.IsOpen, f.IsClosed] (...)
  { f.open() } { f.open(); f.close() }
```

(a) Joining two different typestates. Using `ifDiff` returns an `Either[...]` for later use.

```
if (...) { f.open(); 10 }
    else { f.open(); 20 }
```

```
ifDiff[f.IsClosed, f.IsOpen, f.IsOpen] (...)
  { f.open(); 10 } { f.open(); 20 }
```

(b) While returning other data, using `ifDiff` recovers state transition via `Sigma`-lifting.

```
// next() transitions it.HasMore
//                 to it.HasMore
loop[it.HasMore] (it.hasNext()) {
  val item = it.next()
  /* further computation... */ }
```

```
// nextChecked() transitions it.HasMore
//              to Either[it.HasMore, it.End]
whileLeft[it.HasMore, it.End] {
  val item = it.nextChecked()
  /* further computation... */ }
```

(c) Traversing through an iterator, `it`, using either `loop` or `whileLeft`.

Fig. 9. Examples of using control flow combinators.

Figure 9 compares `ifDiff` to `if`. Without specialized type inference support, we explicitly specify type arguments for `ifDiff`. The first argument is the initial state, the second the state after `tbranch`, and the third the state after `ebranch`. Figure 9a demonstrates a conditional that closes `f: File` in one branch and opens it in another. A regular `if` cannot express this, as the return type would be a union of two different `Sigma` types, eventually `Any`, which is unusable. In contrast, `ifDiff` returns an `Either` capability that can later be consumed by `matchSame`.

Alternatively, Figure 9b performs the same state transition in both branches but returns an `Int`. For `if`, knowledge that a transition occurred is invisible externally because the transition's resulting capability is local to each branch. However, `ifDiff` can recover this information by using `Sigma` lifting to bundle the resulting capability with the returned value.

Figure 9c demonstrates the iteration combinators `loop` and `whileLeft` by traversing through an iterator `it` while tracking typestate. Iterators have two states: `HasMore` indicating more items and `End` marking no more. Retrieving the next item can be done either via `next()`, unsafe and only transitioning from `HasMore` to `HasMore`, or with `nextChecked()`, transitioning to `HasMore` or `End`. Then, `loop` can be used with `next()`, as it requires the same iterator state. On the other hand, `whileLeft` naturally fits `nextChecked()`, repeating `nextChecked()` until it transitions the state to `it.End`. In both cases, `Sigma`-lifting ensures proper capabilities for further computation.

4 Implementation and Discussion

We have implemented a prototype in a fork of the Scala 3 compiler with the experimental capture checker enabled. Our extensions to the capture checker are minor and non-invasive: the existing capture checking test suite (373 tests) continues to pass. The implementation leverages the existing capture checker for capturing types [9, 12, 51] to realize a simplified fragment of reachability types [18], where the distinctions are immaterial. In this section, we discuss the intended safety properties of our approach, along with its theoretical and practical limitations.

4.1 Intended Safety Properties

As this work primarily focuses on language design, a complete formalization integrating reachability types, path-dependent types, implicit resolution, and ANF transformation is beyond the scope of this paper. Nevertheless, we can characterize the intended safety properties by examining the three core capability operations in our framework: *receive*, *revoke*, and *return*.

- **Receive**: a capability can only be received while it is live, *i.e.*, associated with its object via path-dependent type and not yet revoked.
- **Revoke**: once revoked, a capability is permanently unavailable. Following the rules of destructive effects, a revoked capability cannot be used explicitly or summoned via implicit resolution.
- **Return**: a returned capability is inserted into the implicit scope at the call site, then the compiler can automatically supply it to any subsequent operations that require it until revoked.

Although a full soundness proof is beyond the scope of this paper, we empirically validate our approach through the case studies in Section 3. We refer interested readers to our implementation and accompanying examples for details [3].

4.2 Design and Limitations

Our prototype introduces two main extensions to the compiler. The *destructive effect checker* is implemented as a compiler phase directly after capture checking, architected as a bidirectional typer [20, 52]. It re-types the capture-checked syntax tree while recording a set of killed capabilities, promulgating kill annotations to types. The *type-directed ANF transform* [24, 57] is implemented in the Scala typer, triggered when encountering expressions of type `Sigma`. As the transformation lifts

such expressions, preservation of evaluation order is ensured by marking the generated bindings as `lazy`. We discuss the design choices and limitations below.

Simplification on Destructive Effects. Our destructive effect checker draws on Deng et al. [18], which formalizes *use* and *kill* effects on top of System $F_{<:}$ with higher-order references and reachability qualifiers. Our implementation adopts a simplified subset sufficient for modeling capability revocation, which introduces several limitations.

First, we omit the explicit *use* effect and instead conservatively approximate usage via *mentioning* information encoded in qualifiers. While Deng et al. [18] distinguishes between mentioning and actual use, allowing only uses of killed resources to be forbidden, the precision requires latent effect annotations on function types and significantly increases the complexity of the type system, which is unnecessary for our minimalistic extension.

Second, we omit mutable state, thus forbidding destructive effects on mutable variables and object fields. However, objects with typestate can still be stored in mutable variables and undergo state transitions, since revocation targets capabilities rather than objects.

Third, Deng et al. [18] supports *one-shot functions* that consume/kill captured free variables by using explicitly named self-references in latent effects. Our implementation adopts a simplified static notation `FUN` to denote the self-reference at the innermost (most recent) level. This is limited but practical enough for our implementation atop the capture checker, and can be improved by building on a full reachability type checker.

Sigma. The `Sigma` construct serves as the device for returning bundled capabilities. However, it is not directly expressible using reachability types, which lack dependent types and cannot yet track fresh identities within data structures. As a workaround, functions returning `Sigma` should be understood in continuation-passing style, providing capabilities as fresh to the continuation. Accordingly, `Sigma` should be viewed as a transient wrapper that requires immediate unpacking after return: defining a method that returns `Sigma` requires providing a capability, while calling such a method introduces the implicit capability at the call site.

Error Reporting. Our approach encodes typestate using existing language features rather than introducing it as a built-in primitive. As a result, error messages reflect the underlying encoding, which requires some familiarity to interpret, a tradeoff commonly seen in encoding-based approaches [35, 63]. For instance, in our encoding of session types, performing the same action twice on a channel reports the use of a killed variable, which is meaningful once one understands that the first action kills the channel's capability.

More specifically, typestate violations usually reduce to two kinds of errors, effect-system errors, or implicit-search failures. For example:

```
def g(f: File): f.IsOpen^ ?=> Unit =
  f.close()  // Error as f performs state transition
             //   not reflected in method signature
Type Mismatch Error: Found: (c: f.IsOpen^) ?=> Unit @kill(c); Expected: f.IsOpen^ ?=> Unit
```

Here, a typestate transition induces a kill effect not reflected in the method signature, which is then surfaced as a type mismatch at the source position of the body of `g`. Likewise:

```
withFile { (f: File) => (c: f.IsClosed^) =>
  ()  // no implicit f.IsClosed capability for Sigma-lifting
}
Sigma-lifting failed: No given instance of type f.IsClosed^ was found.
```

This example yields an implicit-search failure at the source position of `()`.

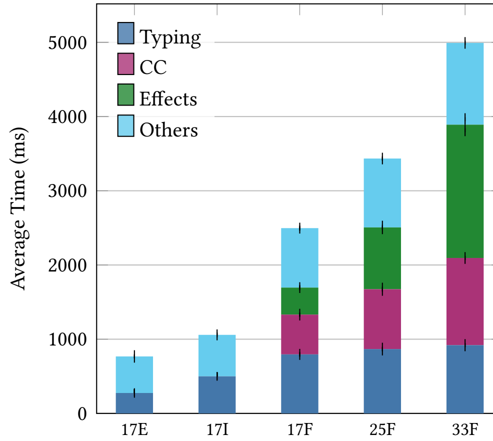


Fig. 10. Compilation time breakdown. Average of 10 runs, measured on macOS Sequoia 15.7.4 (Apple M3 Pro, 18 GB). Black tick above each compilation phase subcategory indicates standard deviation.

While error messages may grow more complex for sophisticated typestates, users need not reason about low-level details such as the ANF transformation of `Sigma`, since source-level metadata (*e.g.*, source positions) can be propagated through `Sigma` to present diagnostics at the appropriate level of abstraction. Moreover, static typestate tracking shifts many errors to compile time, substantially reducing the need for runtime debugging. We expect error reporting can be further improved with additional engineering effort.

Performance Evaluation. We evaluate the performance of our compiler extensions with several benchmarks based on the `DOM` tree API (cf. Figure 3) as it is capable of tracking arbitrarily large typestate using type-level lists. Figure 10 breaks down the compilation time of our benchmarks by phase. All benchmarks are variations on benchmark **17F**, which is a program that contains the code for Figure 5 and creates a `DOM` tree of depth 17. Benchmark **17I** removes `Sigma`, capture checking, and effect checking, only tracking implicit path-dependent capabilities, and benchmark **17E** further removes capabilities. Benchmarks **25F** and **33F** are based on **17F** but create a `DOM` tree tracking 25 and 33 elements, respectively. For medium-sized typestate (tracking 17 and 25 elements), the overhead introduced by our additions (effect checking and the `Sigma`-directed ANF transform) is comparable to the overhead introduced by the capture checker, which is reasonable. For larger typestate (*e.g.*, **33F**), our effect-checking overhead is more visible than that of the capture checker, which can be mitigated by further engineering effort.

5 Related Work

Representing Effects. A generic treatment of sequential (flow-sensitive) effect systems was explored by Gordon [28] using effect quantales, an algebraic structure equipped with a sequencing operator for composing effects in order. Earlier work in higher-order languages primarily addressed flow-insensitive effects. Lucassen and Gifford [43] proposed a polymorphic calculus with effects regarding region-based memory management. Generalized from regions, Henglein et al. [30] introduced a calculus that represents effects with scope tags. Marino and Millstein [44] characterized effects using *check* and *adjust* to manage capabilities. Brachthäuser et al. [12], Lindley et al. [42], Tang et al. [68] further use capabilities or ambient effects to avoid effect polymorphism.

Within the Scala ecosystem, Rytz et al. [58] introduced relative effect polymorphism to alleviate the annotation overhead for effects. Subsequently, Toro and Tanter [69] proposed a gradual effect

system that integrates static and dynamic effect checking. More recently, effect handlers have been realized as a Scala library [13, 14], leveraging capabilities to manage and control effects.

Continuation-passing style (CPS) transformations [16] and monads [21, 22] have well-established connections to effects [15, 72, 73]. Of particular relevance, Rompf et al. [57] introduced a selective CPS transformation to implement a polymorphic calculus with `shift/reset` [15], based on a flow-sensitive effect system inspired by earlier work [4]. In contrast, composing monads is known to be difficult, requiring additional mechanisms [38, 41, 66].

In this work, we model effects through capabilities. The introduction of capabilities is enabled by a selective ANF transformation, while their revocation is managed by a destructive effect system.

Tracking Typestate. Strom and Yemini [65] introduced the concept of typestate, initially assuming a setting where aliasing could be statically resolved, which is generally not feasible in the presence of pointers. To address typestate in the presence of aliasing, subsequent work has employed whole-program analyses [23, 34, 47]. DeLine and Fähndrich [17] proposed modular typestate checking by distinguishing non-aliased objects, thereby enabling typestate enforcement in those cases.

For aliased objects, Bierhoff and Aldrich [7] advanced the use of fractional capabilities in linear reasoning, which underpins the typestate-oriented programming language, Plaid [1, 25]. In addition to state transitions, Plaid requires annotating access permissions of arguments to support modular aliasing reasoning; our approach does not require such annotations. Saffrich et al. [59] further proposed replacing transition annotations with ordered handling of borrows.

Session types [31, 32, 67] exemplify typestate reasoning by enforcing type-safe communication protocols between concurrent processes. Beyond communication, Gay et al. [26] extended session types to specify object protocols, thereby achieving expressiveness comparable to general typestate systems, with restrictions on object aliasing. While session types usually require passing channels for linear reasoning, proposals have been made [61, 62, 70] to enable direct-style programming.

Session types have been implemented in several languages, such as Rust [35, 40], Haskell [55], and Scala [63]; our example is based on the Rust and Haskell implementations. The Scala implementation [63] relies on run-time enforcement of channel linearity.

Linear Types and Fractional Capabilities. Linear logic [27] restricts the structural rules of contraction and weakening, thereby controlling the duplication and disposal of resources. This logical foundation underpins linear type systems [71], in which values must be used exactly once, facilitating safe and predictable management of side effects and mutable state.

Although resources in practical programming are often shared and thus nonlinear, their capabilities [19, 46] can be abstracted and managed linearly. Such capabilities can be made fractional [10] through splitting and rejoining. For example, the type system of Rust [45] permits concurrent reads when write access is disabled; full control is restored once all read borrows have ended.

The integration of substructural reasoning and fractional capabilities is common in typestate analysis and session type systems for regulating shared access. This approach manifests as access permissions in Plaid [25], linear constraints in Linear Haskell [64], and borrows from ordered partial monoids [59, 60]. In this work, we employ capabilities without substructural reasoning, and therefore do not impose multiplicity or ordering constraints on their use.

Scala. Scala is a programming language that integrates object-oriented and functional paradigms, featuring an advanced static type system. Its type system is formalized as the Dependent Object Types (DOT) calculus [2, 56], which enables types to be parameterized by object paths, thereby supporting precise path-dependent reasoning.

Scala additionally provides implicit argument resolution [50], enabling the automatic inference of function parameters based on type information. This feature facilitates capability-based programming [51] by obviating the need for explicit capability passing. To ensure that capabilities do not

escape their intended scopes, proposals have been made to track them as *second-class* values [54, 75] with restricted usages and lifetimes. More recently, the Scala compiler introduced an experimental capture checker [48], which employs descriptive alias tracking to achieve similar safety guarantees.

Descriptive Alias Tracking with Reachability/Capturing Types. Reachability and capturing type systems both aim to track resources in types, using sets of variable names as qualifiers. Motivated by different use cases, these systems primarily diverge in their treatment of unnamed resources. Early proposals for capturing types [9, 12, 51] focused on preventing unintended escape of critical resources, notably capabilities. Polymorphism is ergonomically supported by boxing enclosed resources; unboxing outside the intended scope is disallowed. In contrast, reachability types [6] track escaping resources via self-references. Their original formulation omitted polymorphism, which was subsequently addressed through lightweight polymorphism and explicit quantification [74].

Both reachability and capturing types continue to evolve, concerning tracking separation [74, 76], mechanisms for implementation [37, 78], and semantic denotation [5, 77]. The experimental Scala capture checker [48] has evolved along several lines of research and remains under active development; it has not yet been described end-to-end in the literature. For the purpose of this work, we focus on a common core whose behavior aligns with reachability types.

Within the framework of reachability types, effect systems relative to reachability-tracked values were informally envisioned from the outset, with potential applications including Rust-style ownership transfer and move semantics [6, 74]. However, until recently, reachability-sensitive effect systems had been neither fully formalized nor implemented in a widely-used language. Bracevac et al. [11] investigate compiler optimizations using an integrated type-effect-dependency system. While He et al. [29] address memory management via flow-insensitive scoped allocations with guaranteed lexical deallocation, Bao et al. [5] proved effect safety for a flow-insensitive effect system using logical relations, but without considering deallocation or other destructive effects. Most relevant to our work, Deng et al. [18] formalize flow-sensitive kill effects with sound deallocation, which serves as the foundation for our approach to typestate tracking via revocable capabilities. With the goal of putting theory into practice, and of studying the practical viability of the approach, the present paper supplies a prototype implementation as an extension of the Scala 3 compiler.

6 Conclusion

In this paper, we show that expressive, flow-sensitive typestate tracking is possible with minimal extensions to existing capability-based systems. By decoupling capability lifetimes from lexical scopes and supporting the revocation and implicit returning of capabilities, our approach enables precise and safe management of stateful resources in imperative code. As key supporting mechanisms, our additions include a destructive effect system and a type-directed ANF transformation. The resulting Scala 3 prototype supports a variety of stateful patterns, while maintaining concise and readable code. This work bridges the gap between scoped reasoning and flow-sensitive expressiveness, advancing the safety and ergonomics of stateful programming.

Data Availability Statement

Details of the this work can be found in our extended version [36] and our artifact [3]. Our artifact is also available at https://github.com/TiarkRompf/scala3/tree/artifact.

Acknowledgments

We thank anonymous reviewers for their valuable feedback. This work was supported in part by NSF award 2348334 and an Augusta University faculty startup package, as well as gifts from Meta, Google, Microsoft, and VMware.

References

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

[2] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World (Lecture Notes in Computer Science, Vol. 9600)*. Springer, 249–272.

[3] Anonymous. 2026. *Artifact for "Typestate via Revocable Capabilities"*. doi:10.5281/zenodo.19341757

[4] Kenichi Asai and Yukiyoshi Kameyama. 2007. Polymorphic Delimited Continuations. In *APLAS (Lecture Notes in Computer Science, Vol. 4807)*. Springer, 239–254.

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

[6] Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–32.

[7] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular typestate checking of aliased objects. In *OOPSLA*. ACM, 301–320.

[8] Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. 2022. Type-level programming with match types. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–24.

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

[10] John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS (Lecture Notes in Computer Science)*. Springer, 55–72.

[11] Oliver Bracevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 400–430.

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

[13] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2018. Effect handlers for the masses. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 111:1–111:27.

[14] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *J. Funct. Program.* 30 (2020), e8.

[15] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming*. ACM, 151–160.

[16] Olivier Danvy and Andrzej Filinski. 1992. Representing Control: A Study of the CPS Transformation. *Math. Struct. Comput. Sci.* 2, 4 (1992), 361–391.

[17] Robert DeLine and Manuel Fähndrich. 2004. Typestates for Objects. In *ECOOP (Lecture Notes in Computer Science, Vol. 3086)*. Springer, 465–490.

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

[19] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155.

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

[21] Andrzej Filinski. 1994. Representing Monads. In *POPL*. ACM Press, 446–457.

[22] Andrzej Filinski. 1999. Representing Layered Monads. In *POPL*. ACM, 175–188.

[23] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2 (2008), 9:1–9:34.

[24] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI*. ACM, 237–247.

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

[26] Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. 2010. Modular session types for distributed object-oriented programming. In *POPL*. ACM, 299–312.

[27] Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102.

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

[29] Siyuan He, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2026. When Lifetimes Liberate: A Type System for Arenas with Higher-Order Reachability Tracking. *Proc. ACM Program. Lang.* 10, OOPSLA1 (2026), 1486–1513.

[30] Fritz Henglein, Henning Makholm, and Henning Niss. 2005. Effect Types and Region-Based Memory Management. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Cambridge, Mass.
[31] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR (Lecture Notes in Computer Science)*. Springer, 509–523.
[32] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP (Lecture Notes in Computer Science)*. Springer, 122–138.
[33] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1 (2016), 3:1–3:36.
[34] Mathias Jakobsen, Alice Ravier, and Ornela Dardha. 2021. Papaya: Global Typestate Analysis of Aliased Objects. In *PPDP*. ACM, 19:1–19:13.
[35] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session types for Rust. In *WGP@ICFP*. ACM, 13–22.
[36] Songlin Jia, Craig Liu, Siyuan He, Haotian Deng, Yuyan Bao, and Tiark Rompf. 2026. Typestate via Revocable Capabilities (Extended Version). arXiv:2510.08889 [cs.PL] https://arxiv.org/abs/2510.08889
[37] Songlin Jia, Guannan Wei, Siyuan He, Yuyan Bao, and Tiark Rompf. 2026. Escape with Your Self: Sound and Expressive Bidirectional Typing with Avoidance for Reachability Types. *Proc. ACM Program. Lang.* 10, PLDI (2026).
[38] Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Haskell*. ACM, 94–105.
[39] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly typed heterogeneous collections. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*. ACM, 96–107.
[40] Wen Kokke. 2019. Rusty Variation: Deadlock-free Sessions with Failure in Rust. In *ICE (EPTCS)*. 48–60.
[41] Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *POPL*. ACM Press, 333–343.
[42] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *POPL*. ACM, 500–514.
[43] John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *POPL*. ACM Press, 47–57.
[44] Daniel Marino and Todd D. Millstein. 2009. A generic type-and-effect system. In *TLDI*. ACM, 39–50.
[45] Nicholas D. Matsakis and Felix S. Klock II. 2014. The rust language. In *HILT*. ACM, 103–104.
[46] Mark S. Miller and Jonathan S. Shapiro. 2003. Paradigm Regained: Abstraction Mechanisms for Access Control. In *ASIAN (Lecture Notes in Computer Science, Vol. 2896)*. Springer, 224–242.
[47] Nomair A. Naeem and Ondrej Lhoták. 2008. Typestate-like analysis of multiple interacting objects. In *OOPSLA*. ACM, 347–366.
[48] Martin Odersky et al. 2023. *Scala 3 Reference - Capture Checking*. https://docs.scala-lang.org/scala3/reference/experimental/cc.html
[49] Martin Odersky et al. 2025. *Separation Checking*. https://nightly.scala-lang.org/docs/reference/experimental/capture-checking/separation-checking.html
[50] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2018. Simplicitly: foundations and applications of implicit function types. *Proc. ACM Program. Lang.* 2, POPL (2018), 42:1–42:29.
[51] Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondrej Lhoták. 2021. Safer exceptions for Scala. In *SCALA/SPLASH*. ACM, 1–11.
[52] Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored local type inference. In *POPL*. ACM, 41–53.
[53] Martin Odersky and Matthias Zenger. 2005. Scalable component abstractions. In *OOPSLA*. ACM, 41–57.
[54] 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.
[55] Riccardo Pucella and Jesse A. Tov. 2008. Haskell session types with (almost) no class. In *Haskell*. ACM, 25–36.
[56] Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *OOPSLA*. ACM, 624–641.
[57] 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.
[58] Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *ECOOP (Lecture Notes in Computer Science, Vol. 7313)*. Springer, 258–282.
[59] Hannes Saffrich, Yuki Nishida, and Peter Thiemann. 2024. Law and Order for Typestate with Borrowing. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1475–1503.
[60] Hannes Saffrich, Janek Spaderna, Peter Thiemann, and Vasco T. Vasconcelos. 2025. Borrowing from Session Types. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 3426–3453.
[61] Hannes Saffrich and Peter Thiemann. 2022. Relating Functional and Imperative Session Types. *Log. Methods Comput. Sci.* 18, 3 (2022).
[62] Hannes Saffrich and Peter Thiemann. 2023. Polymorphic Typestate for Session Types. In *PPDP*. ACM, 12:1–12:15.
[63] Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *ECOOP (LIPIcs)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:28.

[64] Arnaud Spiwack, Csongor Kiss, Jean-Philippe Bernardy, Nicolas Wu, and Richard A. Eisenberg. 2022. Linearly qualified types: generic inference for capabilities and uniqueness. *Proc. ACM Program. Lang.* 6, ICFP (2022), 137–164.

[65] Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171.

[66] Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436.

[67] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-based Language and its Typing System. In *PARLE (Lecture Notes in Computer Science)*. Springer, 398–413.

[68] Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2025. Modal Effect Types. *Proc. ACM Program. Lang.* 9, OOPSLA1 (2025), 1130–1157.

[69] Matías Toro and Éric Tanter. 2015. Customizable gradual polymorphic effects for Scala. In *OOPSLA*. ACM, 935–953.

[70] Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. 2006. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.* 368, 1-2 (2006), 64–87.

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

[72] Philip Wadler. 1992. The Essence of Functional Programming. In *POPL*. ACM Press, 1–14.

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

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

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

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

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

[78] Yichen Xu and Martin Odersky. 2023. Formalizing Box Inference for Capture Calculus. *CoRR* abs/2306.06496 (2023).

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