What's in the Box

Ergonomic and Expressive Capture Tracking over Generic Data Structures

YICHEN XU, EPFL, Switzerland
OLIVER BRAČEVAC, EPFL, Switzerland
CAO NGUYEN PHAM, EPFL, Switzerland
MARTIN ODERSKY, EPFL, Switzerland

Capturing types in Scala unify static effect and resource tracking with object capabilities, enabling lightweight effect polymorphism with minimal notational overhead. However, their expressiveness has been insufficient for tracking capabilities embedded in generic data structures, preventing them from scaling to the standard collections library – an essential prerequisite for broader adoption. This limitation stems from the inability to name capabilities within the system's notion of box types.

This paper develops System Capless, a new foundation for capturing types that provides the theoretical basis for reach capabilities (rcaps), a novel mechanism for naming "what's in the box". The calculus refines the universal capability notion into a new scheme with existential and universal capture set quantification. Intuitively, rcaps witness existentially quantified capture sets inside the boxes of generic types in a way that does not require exposing existential capture types in the surface language. We have fully mechanized the formal metatheory of System Capless in Lean, including proofs of type soundness and scope safety. System Capless supports the same lightweight notation of capturing types plus rcaps, as certified by a type-preserving translation, and also enables fully optional explicit capture-set quantification to increase expressiveness.

Finally, we present a full reimplementation of capture checking in Scala 3 based on System Capless and migrate the entire Scala collections library and an asynchronous programming library to evaluate its practicality and ergonomics. Our results demonstrate that reach capabilities enable the adoption of capture checking in production code with minimal changes and minimal-to-zero notational overhead in a vast majority of cases.

CCS Concepts: • **Theory of computation** → **Type structures**; • **Software and its engineering** → *Polymorphism.*

Additional Key Words and Phrases: Scala, Capture Checking, Effect Polymorphism, Generic Data Structures

ACM Reference Format:
Yichen Xu, Oliver Bračevac, 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, Article 334 (October 2025), 28 pages. https://doi.org/10.1145/3763112

1 Introduction

Statically tracking effects and resources through type systems has attracted increasing research efforts in programming languages [25, 15, 10, 9, 47]. Despite this growing body of research, integrating effect systems into mainstream programming languages remains challenging due to concerns about usability and flexibility. Capturing Types (CT)[1] [5] is a promising advancement

[1]We also refer to the approach of Capturing Types as *capture checking* and *capture tracking*.

Authors' Contact Information: Yichen Xu, EPFL, Lausanne, Switzerland, yichen.xu@epfl.ch; Oliver Bračevac, EPFL, Lausanne, Switzerland, oliver.bracevac@epfl.ch; Cao Nguyen Pham, EPFL, Lausanne, Switzerland, nguyen.pham@epfl.ch; Martin Odersky, EPFL, Lausanne, Switzerland, martin.odersky@epfl.ch.

ACM 2475-1421/2025/10-ART334
https://doi.org/10.1145/3763112

that applies the object-capability model [33] to provide a simple, safe, and practical foundation for effect tracking in Scala. Developing an effect system for such an established language brings both opportunities and constraints: while the existing ecosystem facilitates adoption, legacy designs and pre-existing code bases impose strict requirements for ergonomics and backward compatibility.

Bringing Effect Tracking to the Masses. The key to making effect tracking practical and usable in established languages is to describe effect polymorphism without sacrificing flexibility. Since effects are transitive along call edges, every higher-order function needs to be effect-polymorphic to account for effects performed by its arguments. In effect systems with explicit quantifiers, effect parameters in signatures are required. For instance, the familiar `map` function would become the following in a hypothetical effect system with explicit quantifiers:

```
class List[+T] { def map[U, E](f: T -> U eff E): List[U] eff E }
```

While such verbosity is acceptable in principle, especially when designing a new language, it quickly becomes disruptive in established languages like Scala where signature complexity grows rapidly. CT addresses this challenge through lightweight effect signatures and implicit effect polymorphism, a design goal shared by many recent works on other languages [25, 15, 10, 9, 47].

In CT's approach, effects are performed and resources are accessed via capabilities, which are objects that are referenced and tracked as regular program variables. CT's essence is to *track captured variables in types*. It introduces *capturing types* that augment regular types with a *capture set* over-approximating the variables a value may capture, and thus giving a handle on the effects a value may perform. Consider a function that greets someone using console I/O:

```
(name: String) => console.println("Hello␣" + name)
```

Assuming `console` is a capability for console I/O, this function has type `String ->{console} Unit` (shorthand for `(String -> Unit)^{console}`). This *capturing type* consists of (1) the *shape type* specifying parameter and return types, and (2) the *capture set* `{console}` indicating the function may capture `console`. The type makes it evident that the function may perform console I/O.

CT's lightweight notation enables effect polymorphism that is minimally invasive. Returning to our earlier example, the signature of `map` remains *unchanged* from vanilla Scala:

```
class List[+T] { def map[U](f: T => U): List[U] /* no signature changes needed */ }
```

The function arrow `T => U` is shorthand for `(T -> U)^{cap}`, where `{cap}` is the top capture subsuming all capabilities. This allows argument `f` to capture arbitrary capabilities, making `map` effect-polymorphic, analogous to modeling polymorphism through subtyping and `Object` in OO languages.

Retrofitting Capturing Types into Scala. Another key concern is harmoniously and non-invasively integrating the new universe of capturing types (like `String ->{console} Unit`) into the existing Scala type system. Previous work (System $\mathrm{CC}_{<:\square}$ [5]) formally studied a pragmatic and sound way of retrofitting this universe in a model λ-calculus that informed Scala's CT implementation. Consider a set of concurrent futures `Set[Future[T]^]`, where the hat annotation `Future[T]^` (shorthand for `Future[T]^{cap}`) indicates that futures are tracked as capabilities. Under the hood, $\mathrm{CC}_{<:\square}$ requires generic type arguments with captures to be *boxed*, i.e., this type desugars to `Set[box Future[T]^{cap}]`. The appeal of boxing is that existing generic types like `Set[T]` do not need to be polluted with extra quantifiers for captures, and work with normal Scala types as well as capturing types, contributing to the lightweight nature of CT.

Generic Types, Elusive Captures. Attempts to apply CT to Scala's collection library, however, revealed a key limitation of $\mathrm{CC}_{<:\square}$ that makes using generic data structures impractical. Consider a function that turns a set of futures into a stream arranging them by completion order:

```
def collect[T](fs: Set[Future[T]^]): Stream[Future[T]^] =
  val channel = Channel()
  fs.forEach(_.onComplete(v => channel.send(v))) // error, elements inaccessible!
  Stream.of(channel)
```

This function is *untypeable* in $\mathrm{CC}_{<:\square}$! To prevent unsafe capability leaks, the calculus forbids accessing a boxed value when its capture set is the top element `{cap}` [5]. Hence, the futures in `fs` are inaccessible and `collect` does not type-check (we detail boxes and `{cap}` in Section 2). $\mathrm{CC}_{<:\square}$ lacks the ability to handle *nested* captures within generic types.

Furthermore, even if the type system tolerated capability leaks and allowed unboxing `cap`-qualified types, the function would be impractical. The result type `Stream[box Future[T]^{cap}]` is imprecise: it suggests futures in the stream may perform *arbitrary* effects, despite these futures originating from the input `fs` and performing at most the effects of futures in that collection. $\mathrm{CC}_{<:\square}$ cannot express this precise input-output relationship. Is boxing doomed?

The Problem: What's in the Box? The fundamental problem is the lack of a mechanism for safely accessing and referring to capabilities inside boxes. To see the core issue, we approach the problem from the angle of explicit capture quantifications. The universal capture `cap` can be understood as an existential capture. For instance, the type `Future[T]^{cap}` means a future capturing *some* arbitrary capabilities; so it can be viewed as `∃c.Future[T]^{c}`. Under this perspective, the `collect` signature then becomes:

```
def collect[T](fs: Set[box ∃c1.Future[T]^{c1}]): Stream[box ∃c2.Future[T]^{c2}]
```

This reveals the disconnect between parameter and result captures. Since result futures originate from input futures, a more precise and desirable signature would be:

```
def collect[T][∀c1](fs: Set[box Future[T]^{c1}]): Stream[box Future[T]^{c1}]
```

Here, the witness c_1 flows from input to output. Unfortunately, this signature is inexpressible in $\mathrm{CC}_{<:\square}$. The lesson to be learned here is that "no two `cap`s are created equal": we need more granular means to distinguish between them.

As one solution, the CT system we propose supports *optional* explicit capture polymorphism:

```
def collect[T, C^](fs: Set[box Future[T]^{C}]): Stream[box Future[T]^{C}]
```

While this works with `C^` declaring a *capture parameter*, relying on explicit polymorphism alone would undermine CT's lightweight design. From a language design perspective, we would like to keep explicit polymorphism optional and offer ergonomic alternatives for such a common pattern.

Reach Capabilities: Existentials without the Clutter. We propose *reach capabilities* as an effective and lightweight means to name existential captures in boxes. With reach capabilities, the `collect` signature becomes:

```
def collect[T](@use fs: Set[Future[box T]^]): Stream[box Future[T]^{fs*}] // <- precise capture {fs*}
```

This signature tracks the futures in `fs` through: (1) the reach capability `{fs*}` that names *what's in the box* of `fs`'s type; and (2) the `@use` annotation signifies that the reach capability is used by `collect` (see Section 2.2.3). No extra universal quantifiers or existential types are inflicted upon users.

A New Foundation for Capturing Types. Previous attempts at supporting naming mechanisms for box contents in Scala 3 [56] suffered from several soundness issues [43, 44, 23]. This experience, along with our analysis of `cap`'s limitations, motivated us to develop a new theoretical foundation for CT. We present two calculi: **System Capless**, a new foundational capture calculus with explicit universal and existential capture quantification, and **System Reacap**, a surface calculus formalizing CT's lightweight syntax with reach capabilities. System Capless provides the new theoretical bedrock for capture tracking, while System Reacap maintains CT's lightweight design. A type-preserving translation from Reacap to Capless assigns precise meaning to the surface syntax. System Capless also informs our new quantifier-based capture checker implementation for Scala 3.

Scala Collections, Capture Checked. The new and improved capture-checker implementation based on System Capless finally enables integrating capturing types into Scala's entire standard collections library with minimal modifications (<5% LoC changed, almost 90% function signatures stay the same). The kinds of modifications typically look as follows:

```
class Set[T]: // a mutable set
  def filterInPlace(pred: T => Boolean): this.type   // no changes
  def prependAll(items: IterableOnce[T]^): this.type // extra universal capture set `^`
class Iterator[T]:
  def map(f: T => Boolean): Iterator[T]^{this, f}    // capture set {this, f} on return type
```

When the lightweight notation falls short, e.g., for mutable builders (Section 6), we support *optional* explicit capture parameters. Notably, the collections library required *none*! Thus, our work is a decisive step towards bringing practical effect systems to real-world programming languages.

Contributions. To summarize, our contributions are as follows:

- *Reach Capabilities:* We motivate reach capabilities (Section 2), which enable expressive and lightweight effect polymorphism over generic data structures. We explain the subtleties of the previous system's boxing mechanism and universal capability that necessitate reach capabilities.
- *System Capless:* We present a new foundation for capturing types (System Capless, Section 3) with existential and universal quantification of capture sets. Beyond providing the theoretical basis of reach capabilities, it is a more principled and expressive formalization of capture checking compared to the previous system $\mathrm{CC}_{<:\square}$ [5].
- *System Reacap:* We present System Reacap (Section 4) which formalizes the surface language of capture checking with rcaps, whose semantics is defined by a type-preserving translation to System Capless.
- *Mechanized Metatheory:* We establish the type soundness and scope safety of our capture tracking system (Section 5). The metatheory of System Capless is mechanized in Lean 4, while pencil-and-paper proofs of the type-preserving translation from System Reacap to Capless are provided in the supplementary material.
- *Implementation and Evaluation:* We applied the theory in a full re-implementation of Scala 3's capture checker (Section 7) which was used to compile capture-checked versions of an asynchronous programming library (Section 6), and Scala's standard collections library. We assess the required changes to the latter's ~30K-line code base in Section 7. The required change set is simple and small enough to make the transition to capture checking practical.

Finally, we discuss limitations and future directions in Section 8, related work in Section 9 and conclude in Section 10. The Lean 4 mechanization, the compiler implementation and the code for evaluation are available in the accompanying artifact [54].

2 A Tale of Names and Boxes

This section motivates *reach capabilities* and the two proposed calculi. All examples can be compiled by our implementation which is part of the Scala 3 compiler.

2.1 A Brief Introduction to Capture Tracking

Bringing effect tracking to a well-established, mainstream language like Scala poses specific constraints. Scala's broad adoption makes the ecosystem highly sensitive to notational overhead and backward compatibility: systems that demand substantial syntactic changes or disrupt established idioms are unlikely to be viable. Classical type-and-effect systems face a fundamental propagation problem: effects flow along call chains, forcing each function to account for the effects of its callees. To cope, such effect systems typically choose between manual specialization for fixed effect classes (duplication) or pervasive effect annotations (syntactic burden). This combination of propagation and notation has been a major barrier to deploying effect systems in Scala [5].

The capability-based approach circumvents this fundamental issue by modeling effects through capabilities tracked in the type system. Rather than explicitly tracking effect propagation, capabilities naturally flow through the program as ordinary program variables. Nevertheless, capability systems face the problem of *captures* [5], where closures can "leak" effects outside of their designated lifetime by simply holding a reference to such capabilities. Capturing types (CT) [5, 38] takes this capability-oriented approach while proposing a lightweight mechanism to track captures, bringing effective-yet-ergonomic effect tracking to Scala. A capturing type tracks the capabilities a value can capture and takes the form of $T\,\hat{}\,\{x_1, \cdots, x_n\}$, consisting of two components: (1) the **shape type** T, a "classical" type describing the shape of the value (e.g. `Int`, a function from `Int` to `Int`, etc.), and (2) the **capture set** $\{x_1, \cdots, x_n\}$, a set of program variables a value of this type can at most capture. Consider the function below which prints a greeting, using the capability `console` for console I/O:

```
def sayHi(name: String): Unit = console.log(s"Hi,␣$name!")
```

`sayHi` has the capturing type `(String -> Unit)^{console}`. Since capabilities are represented as variables, the capture set indicates the effects and resources a value of this type can produce and access. Here, the capture set of `sayHi` indicates that the function *at most* performs console I/O.

2.1.1 *Lightweight Effect Polymorphism.*

CT supports *implicit effect polymorphism.* For higher-order functions, classical effect systems have to use explicit effect binders to track effects like in the list `map` example from Section 1.

At its core, the issue with effect binders is about *naming*. When writing a higher-order function, a name is needed to account for the effect produced by the argument. Failing to do so leads to either a restrictive or an imprecise type.

By contrast, in CT, the signature of the list `map` method stays unchanged:

```
trait List[+A] { def map[B](f: A => B): List[B] }
```

It is effect-polymorphic yet stays *identical* to the original signature. Under the hood, `A => B` expands to `A ->{cap} B` which itself is a shorthand for `(A -> B)^{cap}`. This type is a function from A to B capturing at most `{cap}` where **`cap`** is the *universal capability.*

2.1.2 *Universal Capability as a Device for Effect Polymorphism.*

In CT, every capability is derived from a set of existing ones, forming a hierarchy of authority. The universal capability **`cap`** is the root of this *capability hierarchy.* E.g., the following `Logger` class writes logging messages to both a file and the console:

```
class Logger(f: File^) { def log(msg: String): Unit = { f.write(msg); console.log(msg) } }
val f: File^ = ...
val logger: Logger^{f,console} = Logger(f)
```

Here, `File^` (short for `File^{cap}`) is a capability for file I/O. The `logger` capability obtains access to the file and the console from existing capabilities `f` and `console`, i.e., deriving from `f` and `console`. Furthermore, the `sayHi` function whose type is `String ->{console} Unit` can be viewed as a capability derived from `console`.

CT introduces *subcapturing*, a subtyping relation between capture sets. It augments set inclusion with the capability hierarchy. A capability is a subcapture of the capabilities it derives from. For instance, the following subcapturing relations hold for the `Logger` example:

```
{} <: {logger} <: {f,console}          {} <: {f} <: {cap}
```

Subtyping between capturing types is defined by the combination of regular subtyping between shape types and subcapturing. Since all capabilities are ultimately derived from the universal capability, any capture set is a subcapture of `{cap}`. Therefore, subcapturing and the universal capability **`cap`** can be used as a device for *effect polymorphism.* Going back to `List.map`, the argument type `A => B` indicates that `map` takes functions performing arbitrary effects: it is effect-polymorphic. This is analogous to using `Any` as the top type for subtype polymorphism.

In fact, the argument `f` of `map` itself is a capability, thus `f` becomes a *name* of its effects. We do not need an extra name! For example, the following function takes an operation and returns an iterator that repeatedly applies that operation:

```
def repeated[T](f: () => T): Iterator[T]^{f} = new Iterator[T]:
  def next(): T = f()
  def hasNext(): Boolean = true
```

The signature reads that, given any operation `f`, the function returns an iterator with `f`'s effects.

2.1.3 *What's in the Box?*

Another challenge in supporting capture tracking in Scala is the interaction between capturing types and generics, which is a fundamental part of functional programming. Consider the following generic function that transforms the first element of a pair:

```
def mapFirst[A, B, C](p: Pair[A, B], f: A => C): Pair[C, B] = Pair(f(p.x), p.y)
```

What should be the signature of this function under CT? Without any restrictions, the type variables `A`, `B`, and `C` could be instantiated to capturing types. This means that the pair `p` could capture arbitrary capabilities through its fields. Consequently, the parameter `p` needs to be annotated with `{cap}`:

```
def mapFirst[A, B, C](p: Pair[A, B]^{cap}, f: A => C): Pair[C, B]^{cap} = ...
```

The resulting `Pair` also needs to be annotated with `{cap}` because there is no account of what `C` captures: it can be anything. This is an unacceptably imprecise signature.

The solution in CT is to keep generic types pure and introduce boxes to recover expressiveness. This simplifies generic functions, as they do not need to account for potential effects from their type arguments. With this restriction, the `mapFirst` function works without any capture annotations:

```
def mapFirst[A, B, C](p: Pair[A, B], f: A => C): Pair[C, B] = ...
```

Intuitively, parametricity ensures that `mapFirst` cannot inspect the contents of the pair, so the captures of the generic field types should be irrelevant to this function.

To handle capturing types in generic contexts, CT introduces *boxes*, which encapsulate impure values as pure ones. To access a boxed value, it needs to be *unboxed*, which "pops out" the captures that were previously hidden. For example:

```
val consoleOps: List[box () ->{console} Int]^{} = List(box () => console.readInt())
val f: () ->{} Boolean = () => consoleOps.isEmpty
val g: () ->{console} Int = () => (unbox consoleOps.head)()
```

Even though elements of `consoleOps` are effectful, they are all boxed and the list `consoleOps` is pure. The function `f` is pure since it does not access the elements of `consoleOps`. Conversely, `g` *does* access the elements of `consoleOps` and unboxes its element, which pops out the captured capabilities hidden by the box. Therefore, `g` captures the previously hidden capability `console`.

As a rule of thumb, whenever we see a capturing type in type-argument position, there is implicitly a box. For instance, the type `List[() => Int]` expands automatically to `List[box () => Int]`. We will nevertheless show boxes in examples for pedagogical reasons. In fact, the Scala 3 compiler implements complete box inference, so boxes are transparent to users [57] and the language does not even have a surface syntax for them.[2]

Now we can see how the `mapFirst` function works with boxed impure values. Consider a pair containing a file capability and an operation:

```
val p: Pair[box File^{f}, box () ->{f} Unit] = ...
val q = mapFirst(p, f => box (new Logger(unbox f))) // : Pair[box Logger^{f}, box () ->{f} Unit]
val useLogger = () => (unbox q.fst).log("test")     // : () ->{f} Unit
```

The generic function `mapFirst` operates on this pair with all the impure values boxed. The transformation function creates a new `Logger` by unboxing the file, and the result is re-boxed to maintain

[2] Unlike the "boxing" in the JVM, boxes in CT are purely a compile-time construct with no runtime overhead.

purity. When the logger is finally used, unboxing it reveals the capability `f` in the capture set. This demonstrates the idea of *capture tunnelling* [5]: captures are tunneled through generic contexts via boxes and only surface when the boxed values are accessed. This reflects the relational parametricity [50] of generic functions, and allows CT to stay concise and practical [5].

Furthermore, boxes play a crucial role in ensuring the scope safety of capabilities: a boxed value capturing **cap** (e.g. **box** `File^{cap}` where `^` binds tighter than the box) cannot be unboxed, as it typically represents a capability that has escaped from its defining scope [5]. The following example tries to leak a `File` out of its defining local scope:

```
def withFile[T](f: File^ => T): T = { val l: File^ = new File; val r = f(l); l.close(); r }
val leakedBox: box File^{cap} = withFile[box File^{cap}](file => box file)
val leakedFile = unbox leakedBox // error: unboxing value capturing {cap}
```

Note the type parameter to `withFile` has to be instantiated with **box** `File^{cap}`, as the local parameter `file` is out of scope. Consequently, the unbox operation fails due to the above restriction.

Despite these properties, boxes introduce a fundamental limitation: they *cut the tie* between the name of a generic data structure and the effects of its elements, making them untrackable.

For instance, given `ops: List[() => Int]`, we cannot use `ops` to name the effects of the list elements:

```
def mkIterator[T](ops: List[() => T]): Iterator[T]^{cap} = ...
```

Here, `mkIterator` creates an iterator from a list of closures, running them one by one. This function cannot be expressed in the previous CT system [5] due to the scope safety restriction mentioned above. Furthermore, even if we had sacrificed scope safety and lifted the restriction, we are only able to type the result at `Iterator[T]^{cap}` since we have no means to *name* the effects of the elements in `ops`. This is again utterly imprecise: even if only pure operations are passed in, the result is considered performing arbitrary effects. The following definition of `pure` has a pure RHS, but will fail to type-check:

```
val pure: () ->{} Int = () => mkIterator(List(() => 1)).next() // error: using value capturing {cap}
```

The root cause lies in the `mkIterator` example itself. The argument `ops` (of type `List[box () => T]^{}`) is pure: `{ops} <: {}`, since list elements are *boxed*. Therefore, the list's capture set becomes completely disconnected from the capture sets of its elements. Hence, we cannot name *what's in the box* of a generic data structure! This inability to characterize the contents of boxes is precisely the underlying problem that this paper addresses.

2.2 Naming What's in the Box

The problem of naming capabilities inside boxes can be solved by extending the type system with explicit quantification over capture sets. This is supported by our new foundational calculus, System Capless, which provides a sound and principled basis for explicit capture quantifications. We can give a precise type to our `mkIterator` example:

```
def mkIteratorExplicit[T, C^](ops: List[() ->{C} T]): Iterator[T]^{C} = ...
```

Here, `[..., C^]` introduces a universal capture set variable `C`. The signature now precisely states that for any capture set `C`, if `mkIterator` is given a list of operations that all at most capture `C`, it returns an iterator that also captures `C`.

```
mkIteratorExplicit[Int, {}](List(() => 1, () => 2)) // : Iterator[Int]^{}
mkIteratorExplicit[Int, {console}](consoleOps)      // : Iterator[Int]^{console}
```

When called with pure operations, `C` can be instantiated to the empty set `{}`, and the resulting iterator is pure. When called with `consoleOps` of type `List[() ->{console} Int]`, `C` is instantiated to `{console}`, and the result captures `{console}`.

While expressive, this explicit style can be verbose: every function that maps generic collections of capabilities has to be annotated with explicit capture variables. This deviates from the lightweight philosophy that makes CT appealing for established languages like Scala. We therefore propose to keep explicit quantification as an optional feature, and introduce *reach capabilities*, an ergonomic and lightweight mechanism for naming "what's in the box". Reach capabilities allow us to write the `mkIterator` signature as follows, with minimal disruption to the original code:

```
def mkIterator[T](@use ops: List[() => T]): Iterator[T]^{ops*} = ...
```

Reach capabilities can be understood by translating them to explicit capture variables. For instance, the reach capability `ops*` directly corresponds to the variable `c` in the explicit version. It serves as a *name* for the capabilities that can be *reach*ed through the boxes of `ops`. Similar to the explicit version, the following is well-typed:

```
mkIterator(List(() => 1, () => 2)) // : Iterator[Int]^{}
mkIterator(consoleOps)             // : Iterator[Int]^{console}
```

Reach capabilities are realized through three core mechanisms: reach refinement (Section 2.2.1), deep capture sets (Section 2.2.2), and the **@use** annotation (Section 2.2.3).

2.2.1 Reach Refinement. To introduce rcaps, the type-checker performs *reach refinement*. When a variable `ops` is used, this process replaces certain occurrences of the universal capability **cap** in its type with the reach capability `ops*`. This essentially gives a name to the capabilities inside the boxes, which is analogous to how a variable names the capabilities it directly captures. For instance, given `ops: List[() ->{cap} T]`, reach refinement infers its type as `List[() ->{ops*} T]`. Let's inspect the `mkIterator` function as an example:

```
def mkIterator[T](@use ops: List[box () => T]): Iterator[T]^{ops*} = new Iterator[T]:
  var current: List[box () ->{ops*} T] = ops
  def next(): T =
    val f: () ->{ops*} T = unbox (current.head : box () ->{ops*} Unit)
    current = current.tail
    f()
  def hasNext(): Boolean = current.nonEmpty
```

This type-checks thanks to reach refinement:

- In the definition of `current`, `ops` is accepted because its type is refined from `List[box () ->{cap} Int]` to `List[box () ->{ops*} Int]`.
- On the RHS of the variable definition `f`, `current.head` has type `box () ->{ops*} Int`.
- The unboxing then propagates the reach capability `ops*` to the capture set of the iterator's closure.
- The resulting iterator correctly captures `ops*`.

From an explicit-quantification perspective, the parameter `ops` has the type $\exists c.$ `List[box () ->{`c`} T]`, and `ops*` corresponds directly to the witness of c.

Understanding rcaps in terms of a translation to quantification is essential for a sound design. The earlier, ad-hoc implementation of rcaps suffered from soundness issues precisely because it lacked this foundation [43, 44, 23]. The central question is: which occurrences of the universal capability **cap** in a type should be refined to a reach capability? The initial, intuitive answer was "all covariantly-occurring **caps**". This is unsound, accepting the following code:

```
val map: [T] -> (files: List[box File^]) -> (op: (box File^) => T) -> List[T] =
  files.map(op) // a function that maps a list of files
val makeFilePure: File^ -> File^{map*} = (f: File^) => map[box File^{map*}](List(f))(x => x).head
```

The `makeFilePure` function is problematic: it converts an arbitrary `File^` capability (which can perform file I/O) to one that only captures `map*`. Since `map` is a pure function, `map*` is empty, meaning the resulting `File` is considered *pure*, despite being the same impure `File` that was given!

The problem lies within `map`'s reach refinement:

```
[T] -> (files: List[box File^{cap}]) -> (op: (f: box File^{map*}) => T) -> T
```

It becomes clear that this is absurd when we translate the type of `map` into explicit quantification:[3]

```
[T] -> (files: ∃c1. List[box File^{c1}]) -> (op: (f: ∃c2. box File^{c2}) => T) -> T
```

`map*` witnesses the existential scoped at the outermost level of `map` (in this case, there is no existential bound at this level), and clearly does not witness c_2. The unsound reach refinement principle implicitly assumes `map`'s type translates to:

```
∃c2. [T] -> (files: ∃c1. List[box File^{c1}]) -> (op: (f: box File^{c2}) => T) -> T
```

It confuses the scope of the existential variable c_2. Our principled approach avoids such confusion by ensuring that each **cap** is mapped to an existential in the closest enclosing scope, and reach capabilities always witness the outermost existential.

2.2.2 Deep Capture Sets. At function call sites, reach capabilities of parameters are instantiated with the *deep capture sets* of the corresponding argument types. The deep capture set of a type collects all capabilities that occur covariantly in that type. It provides a concrete witness for the existential quantification that reach capabilities represent.

Let us revisit our `mkIterator` example with its explicit quantification version:

```
def mkIteratorExplicit[T, c^](ops: List[() ->{c} T]): Iterator[T]^{c} = ...
```

When calling `mkIteratorExplicit[Int, {console}](consoleOps)`, the capture variable `c` is explicitly instantiated to `{console}`. Deep capture sets achieve the same for reach capabilities: when calling `mkIterator(consoleOps)`, the reach capability `ops*` (which witnesses the explicit variable `c`) is instantiated with the deep capture set of `List[() ->{console} Int]`, which yields `{console}`, correctly typing the result as `Iterator[Int]^{console}`. Deep capture sets collect only covariantly-occurring capture sets because these represent capabilities that can flow "outward" – the capabilities "in the box" that can be accessed when using a value. Contravariant positions, by contrast, represent capabilities flowing "inward" – requirements for using the value rather than capabilities it captures.

*2.2.3 The **@use** Annotation.* The **@use** annotation on a function parameter signifies that the parameter's reach capability is *used* by the function. This annotation is needed to ensure that function applications are tracked correctly. Normally, the capabilities captured by an application `f(x)` are simply `{f,x}`. However, this is unsound for functions that use their parameters' reach capabilities:

```
def runOps(@use ops: List[() => Unit]): Unit = ops.foreach(op => op()) // run each op in list
val ops: List[() ->{console} Unit] = List(() => console.log("Hello"))
val r2 = () => runOps(ops)
```

Without a special rule for **@use** parameters, the body of `r2` would only capture `{runOps, ops}`, which is pure. As a result, `r2` would be incorrectly typed as `() ->{} Unit`, even though it performs console I/O. The **@use** annotation signals that the call site must account for the capabilities *inside* the argument. The general rule is that for **@use** parameters, the call captures the deep capture set of the argument, whereas for normal parameters only its (shallow) capture set is used. For `runOps(ops)`, the captured set is the union of `{runOps}` and the deep capture set of `ops`'s type, which is `{console}`. This correctly gives `r2` the type `() ->{console} Unit`.

2.2.4 Type Definitions. Sometimes, the default behavior of introducing existential quantifiers in the closest enclosing scope is not desirable. To address this, System Reacap includes *type definitions*, which are essentially parameterized aliases for types. For instance, the following type definition defines non-dependent functions, i.e., those that do not depend on their parameters:

```
type Function[-A, +B] = (z: A) -> B
```

Instances of **cap** in a type definition's type parameters are translated into existentials *before* the type definition is expanded. Consider the `flatMap` function for `Iterator` in the Scala collections library:

[3]For clarity, we show existential quantifiers on parameters, which are trivially convertible into universal quantifiers.

$x,\ y,\ z$	**Variable**
X	**Type Variable**
c	**Capture Variable**
$s,\ t,\ u ::=$	**Term**
a	answer
$x\ y$	app.
$x[S]$	type app.
$x[c]$	capture app.
let $x = t$ in u	let
let $\langle c, x\rangle = t$ in u	existential let
$v ::=$	**Value**
$\lambda(x : T)t$	term lambda

$\lambda[X <: S]t$	type lambda
$\lambda[c <: B]t$	capt. lambda
$\langle C, x\rangle$	pack
$E,\ F ::=$	**Existential Type**
$\exists c.\, T$	existential
T	type
$R,\ S ::=$	**Shape Type**
$\top$	top
X	type variable
$\forall(x : T)E$	term function
$\forall[X <: S]E$	type function
$\forall[c <: B]E$	capt. function

$a ::= x \mid v$	**Answer**
$\theta ::= x \mid c$	**Capture**
$C,\ D ::= \{\theta_1, \cdots, \theta_n\}$	**Capture Set**
$B ::= * \mid C$	**Capture Bound**
$T,\ U ::=$	**Type**
$S\hat{}C$	capturing
S	pure
$\Gamma,\ \Delta ::=$	**Context**
$\emptyset$	empty
$\Gamma, x : T$	term binding
$\Gamma, X <: S$	type binding
$\Gamma, c <: B$	capt. binding

Fig. 1. Abstract syntax of System Capless. Key differences from System $\mathrm{CC}_{<:\Box}$ are highlighted.

```
def flatMap[A, B](it: Iterator[A]^, f: A => Iterator[B]^): Iterator[B]^{?} // <- what's the result?
```

With the default translation scheme, the resulting iterator's capture set is the overly imprecise `{cap}`. This is because type parameter `f`'s type translates to `A =>` $\exists c.$ `Iterator[B]^{c}`. Each application of `f` yields an iterator that captures a locally-quantified existential c. A more desirable translation would be $\exists c.$ `A => Iterator[C]^{c}`, with the existential c being scoped over the whole function, and the reach capability `f*` would be the capture set of the result iterator. By treating non-dependent function types `T => U` as applied type definitions `Function[T, U]^`, we can type-check the following signature with a precise result capture set:

```
def flatMap[A, B](it: Iterator[A]^, f: A => Iterator[B]^): Iterator[B]^{it, f, f*}
```

This is exactly how the implementation works. The applied type `Function[A, Iterator[B]^]` translates and dealiases as follows:

```
Function[A, Iterator[B]^{cap}] ⇝ ∃c. Function[A, Iterator[B]^{c}] ⇝ ∃c. (z: A) -> Iterator[B]^{c}
```

Type definitions play a crucial role in our system by offering a way to change the scope of existentials, as needed by, e.g., church-encoded data structures. We further discuss them in Section 4.2.4.

While explicit quantification over capabilities provides a more powerful and general solution to capability polymorphism, this generality often comes at the cost of verbosity and boilerplate. Reach capabilities, in contrast, offer a lightweight and ergonomic alternative that is sufficient for the vast majority of common programming patterns. For instance, rcaps are sufficiently expressive for capture-checking the standard library. We do not need any explicit quantification. The usefulness of reach capabilities is further demonstrated in the case study on asynchronous programming (Section 6) and in our technical report [55, Appendix B.1].

3 System Capless: A New Foundation for Expressive Capture Tracking

System Capless (Figures 1 and 2) is a new foundation of capturing types. It models the essence of our new capture checker implementation in Scala 3 (Section 7). Formally, it is a version of System $\mathrm{CC}_{<:\Box}$ [5], the main difference being (1) the removal of the universal capability **cap**, and (2) having explicit capture quantifications.

Typing $C;\Gamma \vdash t : E$

$$\frac{x : S\,\hat{}\,C \in \Gamma}{\{x\};\Gamma \vdash x : S\,\hat{}\,\{x\}}\;(\textsc{var}) \qquad \frac{C';\Gamma \vdash x : (\forall(z : T)E)\,\hat{}\,C \quad C';\Gamma \vdash y : T}{C';\Gamma \vdash x\,y : [z := y]E}\;(\textsc{app}) \qquad \frac{C';\Gamma \vdash x : (\forall[c <: D]E)\,\hat{}\,C}{C';\Gamma \vdash x[D] : [c := D]E}\;(\textsc{capp})$$

$$\frac{C';\Gamma \vdash x : [c := C]T}{\{\};\Gamma \vdash \langle C, x\rangle : \exists c.\,T}\;(\textsc{pack}) \qquad \frac{C;(\Gamma, X <: S) \vdash t : E \qquad \Gamma \vdash S \text{ wf}}{\{\};\Gamma \vdash \lambda[X <: S]t : (\forall[X <: S]E)\,\hat{}\,C}\;(\textsc{tabs}) \qquad \frac{C;\Gamma \vdash t : T \qquad C;(\Gamma, x : T) \vdash u : E \qquad \Gamma \vdash C, E \text{ wf}}{C;\Gamma \vdash \text{let } x = t \text{ in } u : E}\;(\textsc{let})$$

$$\frac{C;\Gamma \vdash t : E \qquad \Gamma \vdash E <: F \qquad \Gamma \vdash C <: C' \qquad \Gamma \vdash C', F \text{ wf}}{C';\Gamma \vdash t : F}\;(\textsc{sub}) \qquad \frac{C';\Gamma \vdash x : (\forall[X <: S]E)\,\hat{}\,C}{C';\Gamma \vdash x[S] : [X := S]E}\;(\textsc{tapp})$$

$$\frac{C;(\Gamma, x : T) \vdash t : E \qquad \Gamma \vdash T \text{ wf}}{\{\};\Gamma \vdash \lambda(x : T)t : (\forall(x : T)E)\,\hat{}\,(C \setminus x)}\;(\textsc{abs}) \qquad \frac{C;(\Gamma, c <: B) \vdash t : E \qquad \Gamma \vdash C \text{ wf}}{\{\};\Gamma \vdash \lambda[c <: B]t : (\forall[c <: B]E)\,\hat{}\,C}\;(\textsc{cabs})$$

$$\frac{C;\Gamma \vdash t : \exists c.\,T \qquad C;(\Gamma, c <: *, x : T) \vdash u : F \qquad \Gamma \vdash C, F \text{ wf}}{C;\Gamma \vdash \text{let } \langle c, x\rangle = t \text{ in } u : F}\;(\textsc{let-e})$$

Subcapturing $\Gamma \vdash C_1 <: C_2$

$$\frac{\Gamma \vdash C_1 <: C_2 \qquad \Gamma \vdash C_2 <: C_3}{\Gamma \vdash C_1 <: C_3}\;(\textsc{sc-trans}) \qquad \frac{x : S\,\hat{}\,C \in \Gamma}{\Gamma \vdash \{x\} <: C}\;(\textsc{sc-var}) \qquad \frac{c <: C \in \Gamma}{\Gamma \vdash \{c\} <: C}\;(\textsc{sc-bound}) \qquad \frac{C_1 \subseteq C_2}{\Gamma \vdash C_1 <: C_2}\;(\textsc{sc-elem}) \qquad \frac{\Gamma \vdash C_1 <: C \qquad \Gamma \vdash C_2 <: C}{\Gamma \vdash C_1 \cup C_2 <: C}\;(\textsc{sc-set})$$

Bound Subtyping $\Gamma \vdash B_1 <: B_2$ same as subcapturing plus $\Gamma \vdash B <: *$

Subtyping $\Gamma \vdash E_1 <: E_2$

$$\Gamma \vdash S <: \top\;(\textsc{top}) \qquad \Gamma \vdash E <: E\;(\textsc{refl}) \qquad \frac{\Gamma \vdash E_1 <: E_2 \qquad \Gamma \vdash E_2 <: E_3}{\Gamma \vdash E_1 <: E_3}\;(\textsc{trans}) \qquad \frac{X <: S \in \Gamma}{\Gamma \vdash X <: S}\;(\textsc{tvar}) \qquad \frac{\Gamma \vdash S_1 <: S_2 \qquad \Gamma \vdash C_1 <: C_2}{\Gamma \vdash S_1\,\hat{}\,C_1 <: S_2\,\hat{}\,C_2}\;(\textsc{capt})$$

$$\frac{(\Gamma, c <: *) \vdash T_1 <: T_2}{\Gamma \vdash \exists c.\,T_1 <: \exists c.\,T_2}\;(\textsc{exist}) \qquad \frac{(\Gamma, X <: S_2) \vdash E_1 <: E_2 \qquad \Gamma \vdash S_2 <: S_1}{\Gamma \vdash \forall[X <: S_1]E_1 <: \forall[X <: S_2]E_2}\;(\textsc{tfun})$$

$$\frac{(\Gamma, x : T_2) \vdash E_1 <: E_2 \qquad \Gamma \vdash T_2 <: T_1}{\Gamma \vdash \forall(x : T_1)E_1 <: \forall(x : T_2)E_2}\;(\textsc{fun}) \qquad \frac{(\Gamma, c <: B_2) \vdash E_1 <: E_2 \qquad \Gamma \vdash B_2 <: B_1}{\Gamma \vdash \forall[c <: B_1]E_1 <: \forall[c <: B_2]E_2}\;(\textsc{cfun})$$

Fig. 2. Typing rules of System Capless.

3.1 Syntax

The syntax of System Capless is shown in Figure 1, highlighting key differences to $\mathrm{CC}_{<:\square}$ [5]. Just like $\mathrm{CC}_{<:\square}$, we represent programs in monadic normalform (MNF) [21]. This has the advantage that substitutions in dependent applications are always variable renamings and preserve the structure of types. The main difference stems from dropping the top capture set **cap** in favor of explicit bounded universal and existential capture quantification, which behave similarly to the respective quantifiers in System $\mathrm{F}_{\leq}$ for types. Accordingly, capture sets C can now also mention capture variables c next to term variables x.

Capture quantification is bounded by a capture bound B which can be either a concrete capture set upper bound or *unbounded* (denoted by $*$), though note that there is no "top" capture set any longer. When the bound B is omitted, it defaults to $*$.

Existential capture quantification is unbounded and second class, i.e., confined to the top level in the type system and function result types. The typing judgment assigns types from the syntax category E to terms. The argument types of term functions only range over the syntax category

T. This restriction is a deliberate design choice aimed at the minimality of the system. It does not compromise expressiveness, as functions that would otherwise take existential types as arguments can always be equivalently expressed using universal capture quantification followed by ordinary term functions. Concretely, the type $\forall(z\colon \exists c.\, T)E$ is equivalently represented as $\forall[c]\forall(z : T)E$.

3.2 Type System

The typing judgment $C; \Gamma \vdash t : E$ in Figure 2 states that term t has existential type E under context Γ with use set C. Intuitively, the use set C are the set of capabilities that will *at most* be *used* by the evaluation of term t. The use set of all values is empty, as they are already evaluated. Tracking use sets in the judgment improves over the previous System $\mathrm{CC}_{<:\square}$ [5] in two important ways: (1) capture tracking for curried functions behaves more precisely akin to effect systems, i.e., captures of subsequent function arrows do not accumulate on the current one, and (2) a more streamlined handling of let bindings. We explain them in detail when discussing corresponding typing rules.

Typing rules (Figure 2) are mostly identical to System $\mathrm{CC}_{<:\square}$ with the main difference that typing potentially assigns an existential type E and that there are changes and additions due to the reformulation with use sets and the introduction of universal and existential capture quantifications.

To type a variable x in context (VAR), it has to be included in the use set. Just like in $\mathrm{CC}_{<:\square}$, the capture set is refined to $\{x\}$ in the assigned type and the capture set C in the environment can be recovered via subcapturing. The subtyping rule (SUB) allows refining both the use set and the type.

Type abstraction (TABS) and type application (TAPP) is mostly standard, and follows System $\mathrm{F}_{\leq}$, again remarking that values are pure, having an empty use set. Following $\mathrm{CC}_{<:\square}$, type application takes pure types without capture qualifiers. The rules for universal (CABS), (CAPP) and existential capture quantification (PACK), (LET-E) are in the spirit of System $\mathrm{F}_{\leq}$ and hence unsurprising.

Introducing and eliminating a λ-abstraction in (ABS) and (APP) is for the most part standard. In terms of captures, a λ is pure, i.e., having an empty use set, and the use set of the body t is annotated to the function type. Thanks to the tracking of use sets in the typing judgment, the capture tracking of curried functions is more precise compared to [5]. For instance, consider the term $\lambda(x_1 : \mathsf{Unit}).\ \mathsf{let}\ z_0 = \mathsf{logger.log}(\cdots)\ \mathsf{in}\ \lambda(x_2 : \mathsf{Unit}).\,\mathsf{console.readInt}()$. This term has type $(\forall(x_1 : \mathsf{Unit})(\forall(x_2 : \mathsf{Unit})\mathsf{Int})\wedge\{\mathsf{console}\})\wedge\{\mathsf{logger}\}$, while in the previous system [5] it would have been $(\forall(x_1 : \mathsf{Unit})(\forall(x_2 : \mathsf{Unit})\mathsf{Int})\wedge\{\mathsf{console}\})\wedge\{\mathsf{console}, \mathsf{logger}\}$. Our system has a better account of *when* the captured capabilities are used.

3.2.1 Subtyping and Subcapturing. Subcapturing and subtyping rules (Figure 2) follow System $\mathrm{CC}_{<:\square}$. The former is a preorder on capture sets that subsumes set inclusion, plus the more interesting rule (SC-VAR) which "*reflects an essential property of object capabilities*" [5], namely that the singleton capture set $\{x\}$ refines/derives from the capabilities C from which x was created. For the most part, subtyping integrates subcapturing with the standard subtyping rules for kernel System $\mathrm{F}_{\leq}$ unsurprisingly. In addition, the rules (EXIST) and (CFUN) are for the new quantification forms.

3.2.2 Let Bindings and Avoidance. Let bindings (LET) are typed in a standard manner. Due to dependent typing, we need to avoid mentions of the bound variable in the result type, which we enforce by subtyping. Well-formedness of capture sets $\Gamma \vdash C$ wf and its lifting to types $\Gamma \vdash E$ wf (defined in [55, Appendix A.1]) enforce that all mentioned variables are in scope. Note that the locally bound variable x is not dropped from the use set C. Instead, it needs to be avoided in the use set with subcapturing. This, along with preciser curried function types, enables reasoning about *when* the captured capabilities are used. For instance, assuming $x_f : (\forall(x : \mathsf{Unit})(\forall(y : \mathsf{Unit})\mathsf{Int})\wedge\{\mathsf{console}\})\wedge\{\mathsf{logger}\}$ and unit a constant of type Unit, the term $\mathsf{let}\ z = x_f\ \mathsf{unit}\ \mathsf{in}\ z\ \mathsf{unit}$ has the use set $\{\mathsf{console}, \mathsf{logger}\}$. This is because z, whose type is $(\forall(y : \mathsf{Unit})\mathsf{Int})\wedge\{\mathsf{console}\}$, is mentioned in the body of the let binding. To avoid the locally-bound z in the use set, we must widen z to

$x,\ y,\ z,\ \mathbf{cap}$	**Variable**	$\lambda[c]t$	capture function	$\theta \coloneqq x \mid x^*$	**Capture**
$X,\ Y,\ Z$	**Type Variable**	$\square\, x$	box	$a \coloneqq x \mid v$	**Answer**
κ	**Typedef Name**	$C,\ D \coloneqq \{\theta_1, \cdots, \theta_n\}$	**Capture Set**	$\nu \coloneqq + \mid -$	**Variance**
$s,\ t,\ u \coloneqq$	**Term**	$R,\ S \coloneqq$	**Shape Type**	$T,\ U \coloneqq$	**Type**
a	answer	$\top$	top	$S\,\hat{}\,C$	capturing
$x\ y$	application	X	type variable	S	pure
$x[S]$	type application	$\forall^{\alpha}(x : T)U$	function	$\Gamma,\ \Delta \coloneqq$	**Context**
$x[C]$	capt. application	$\forall[X <: \top]T$	type function	$\emptyset$	empty
let $x = t$ in u	let	$\forall[c]T$	capture function	$\Gamma, x : T$	term binding
$C \multimap x$	unbox	$\square\, T$	boxed	$\Gamma, X <: \top$	type binding
$v \coloneqq$	**Value**	$\kappa[T_1, \cdots, T_n]$	applied type	Γ, c	capture binding
$\lambda^{\alpha}(x : T)t$	function	$\alpha \coloneqq \epsilon \mid \bullet$	**Use Annotation**	$\Theta \coloneqq$	**Typedef Context**
$\lambda[X <: \top]t$	type func.	$d \coloneqq (X_1^{\nu_1}, \cdots, X_n^{\nu_n}) \mapsto S$		$\emptyset$	empty
			Type Definition	$\Theta, \kappa = d$	typedef

Fig. 3. Abstract syntax of System Reacap.

{console}. Conversely, the following term, which applies x_f once and discards the result, will only have the use set {logger}: let $z = x_f$ unit in unit. In the previous system [5], both terms capture {console, logger} though the latter did not use console. Besides, the premises and the conclusion share the same use set C, which is more uniform and streamlined. One can always find a use set that accounts for all the capabilities used by the premises and use (SUB) to make this rule applicable.

Beyond providing a theoretical basis for rcaps, System Capless itself is a more principled and expressive foundation for capture tracking that offers additional theoretical advantages, which are further discussed in our report [55, Appendix B.2]. The evaluation rules are almost identical to System $\mathrm{CC}_{<:\square}$, and we elide them for brevity. See [55, Appendix A.2] for the full details.

4 System Reacap: A Surface-Language Calculus for Reach Capabilities

System Reacap is a core calculus that models the surface language of capture checking with `caps` and reach capabilities. It is built on the previous System $\mathrm{CC}_{<:\square}$ by Boruch-Gruszecki et al. [5]. We support the same lightweight end-user notation in types and extend it with `@use` parameters and reach capabilities `x*` motivated in Section 2. The semantics of Reacap is defined in terms of a type-preserving translation to System Capless with explicit capture quantifiers (Sections 3 and 5).

4.1 Syntax

Figure 3 shows the syntax of System Reacap. The syntax is close to that of System $\mathrm{CC}_{<:\square}$ [5], basically System $\mathrm{F}_{\leq}$ with captures and boxing. The main difference is the addition of reach capabilities x^*, the use-annotation α, and applied types $\kappa[T_1, \cdots, T_n]$. Following System Capless and $\mathrm{CC}_{<:\square}$, terms are in monadic normal form (MNF) [21], i.e., arguments to operations are always let-bound variables. In the formal syntax, the presence/absence of `@use` annotations on parameters is encoded by annotations α, e.g., `(@use x: box (File^C)) ->{D} Int` becomes $(\forall^{\bullet}(x : \square\,(\mathsf{File}\,\hat{}\,C))\mathsf{Int})\,\hat{}\,D$.

Type abstractions take pure type arguments, i.e., type parameters are not qualified with captures and qualified type arguments must be put into boxes. Boxing plays a crucial role in enforcing the scope-safety of capabilities [5]. Type parameters are also unbounded (i.e., always being bounded by $\top$). This choice arises from the translation of System Reacap into Capless (cf. Section 4.3).

An applied type $\kappa[T_1, \cdots, T_n]$ applies a type definition κ to the type arguments. A global type definition context Θ that contains a list of type definitions is assumed. In other words, System Reacap is parameterized by a type definition context Θ. A type definition $\kappa = (X_1^{\nu_1}, \cdots, X_n^{\nu_n}) \mapsto S$ acts like a type macro that, given a list of type arguments $T_1, \cdots, T_n$, expands into the type $[X_1 := T_1, \cdots, X_n := T_n]S$. ν_i is the variance of the i-th type argument, which can be either $+$ (covariant) or $-$ (contravariant). Type definitions enable control over reach-capability expansion (cf. Section 2.2.4) and patterns like church-encoded data types in System Reacap. Their interaction with reach refinement will be discussed in Section 4.2.4.

4.2 Type System

The typing judgement $C; \Gamma \vdash t : T$ in Figure 4 is formulated with use sets like System Capless (Section 3). Rule (VAR) mostly matches the one in System Capless. The additional *reach refinement* (cf. Section 2.2) of the variable's assumed type S to S', which replaces certain occurrences of **cap** in S with the reach capability x^*. We explain the details of reach refinement later in Section 4.2.3.

Rules governing box introduction (BOX) and box elimination (UNBOX) follow $\mathrm{CC}_{<:\square}$. Boxes are considered to be pure values, thus they have an empty use set, and the unboxing operation is only allowed in a context where the use set matches the box's capabilities, as before.

The introduction and elimination of type abstractions (TABS) and (TAPP) are standard. **cap** is not allowed in the deep capture set of the type argument S' in (TAPP), as it leads to ambiguity in the meaning of **cap**s and breaks the translation from Reacap to Capless (cf. Section 4.3).

Let bindings (LET) are typed in a standard manner. The mention of locally bound variable x has to be avoided in the use set C and the result type U by subcapturing and subtyping, respectively. We discuss the more interesting typing rules for abstraction and application next.

4.2.1 *Abstraction and Application.*

The (ABS) rule mostly follows that of System Capless. A function that does not declare its parameter x as used ($\alpha = \epsilon$) is accordingly barred from using the associated reach capability x^* in the use set C of the body.

Dependent function application (APP) resolves reach capabilities. The reach capability of a function with use-parameter ($\alpha = \bullet$) is substituted with the call-site argument's *deep capture set*:

Definition 4.1 (Deep Capture Set). *The deep capture set of a type T under context Γ, denoted as $dcs(\Gamma, T)$, is defined as follows:*

$$\begin{aligned}
dcs(\Gamma, \top) &= \{\} & dcs(\Gamma, \forall[X <: S]T) &= dcs(\Gamma, T)\\
dcs(\Gamma, X) &= dcs(\Gamma, S) \quad \text{if } X <: S \in \Gamma & dcs(\Gamma, \square\, T) &= dcs(\Gamma, T)\\
dcs(\Gamma, \forall^{\alpha}(z : T)U) &= dcs(\Gamma, U) \setminus \{z, z^*\} & dcs(\Gamma, S\,\hat{}\,C) &= dcs(\Gamma, S) \cup C\\
& & dcs(\Gamma, \forall[c]T) &= dcs(\Gamma, T) \setminus \{c\}\\
dcs(\Gamma, \kappa[T_1, \cdots, T_n]) &= \textstyle\bigcup_{X_i^+} dcs(\Gamma, T_i) \quad \text{if } \kappa = (X_1^{\nu_1}, \cdots, X_n^{\nu_n}) \mapsto S \in \Theta
\end{aligned}$$

The deep capture set notion $\mathrm{dcs}(\Gamma, T)$ formally answers "what's in the box?" of a boxed capture type. Our definition generalizes to a richer language like Scala with generic type constructors, e.g., $\mathrm{dcs}(\Gamma, \mathsf{List}[S]) = \mathrm{dcs}(\Gamma, S)$. Variance-aware substitution $[x^* :=_\nu C]$ replaces covariant occurrences of x^* with C and contravariant ones with $\{\}$. ν can be either $+$ (covariant) or $-$ (contravariant). The starting variance is $+$. This ensures the subtyping relation is preserved under substitution.

4.2.2 *Subcapturing and Subtyping.*

The subsumption rule (SUB) allows refining the use set C to C' and the type T to T'. It can only be applied on an answer a, which is either a value or a variable due to the translation mechanism from Reacap to Capless, which will be discussed in Section 4.3.

Like System Capless, subcapturing and subtyping follows those of System $\mathrm{CC}_{<:\square}$ [5], which are mostly unsurprising. We define an ordering $\preceq$ on annotations defined by $\alpha \preceq \alpha$ and $\epsilon \preceq \bullet$, i.e., non-use functions can pass for use functions, but not vice versa (as in (FUN)). The rules (APPLIED-P)

Typing $C; \Gamma \vdash t : T$

$$\frac{x : S \,\hat{}\, C \in \Gamma \qquad \{x^*\} \vdash S \rightsquigarrow S'}{\{x\}; \Gamma \vdash x : S' \,\hat{}\, \{x\}} \ (\textsc{var}) \qquad \frac{C'; \Gamma \vdash a : T' \qquad \Gamma \vdash C' <: C \qquad \Gamma \vdash T' <: T \qquad \Gamma \vdash C, T \text{ wf}}{C; \Gamma \vdash a : T} \ (\textsc{sub})$$

$$\frac{C'; \Gamma \vdash x : S \,\hat{}\, C}{\{\}; \Gamma \vdash \Box x : \Box\, (S \,\hat{}\, C)} \ (\textsc{box}) \qquad \frac{C; \Gamma \vdash x : \Box\, (S \,\hat{}\, C)}{C; \Gamma \vdash C \multimapinv x : S \,\hat{}\, C} \ (\textsc{unbox})$$

$$\frac{C; \Gamma, x : T \vdash t : U \qquad \Gamma \vdash T \text{ wf} \qquad x^* \notin C \text{ if } \alpha = \epsilon}{\{\}; \Gamma \vdash \lambda^\alpha(x : T)t : (\forall^\alpha(x : T)U) \,\hat{}\, (C \setminus \{x, x^*\})} \ (\textsc{abs})$$

$$\frac{C'; \Gamma \vdash x : (\forall^\alpha(z : T)U) \,\hat{}\, C \qquad C'; \Gamma \vdash y : S \,\hat{}\, D \qquad \Gamma \vdash S \,\hat{}\, D <: T \qquad \Gamma \vdash \mathrm{dcs}(\Gamma, S) <: C' \text{ if } \alpha = \bullet}{C'; \Gamma \vdash x\ y : [z^* :=_+ \mathrm{dcs}(\Gamma, S)][z := y]U} \ (\textsc{app})$$

$$\frac{C; \Gamma, c \vdash \lambda[c]t : T \qquad \Gamma \vdash C \text{ wf}}{\{\}; \Gamma \vdash \lambda[c]t : (\forall[c]T) \,\hat{}\, C} \ (\textsc{cabs}) \qquad \frac{C; \Gamma \vdash x : (\forall[c]T) \,\hat{}\, C' \qquad \Gamma \vdash D \text{ wf}}{C; \Gamma \vdash x[D] : [c := D]T} \ (\textsc{capp})$$

$$\frac{C; \Gamma, X <: \top \vdash t : T}{\{\}; \Gamma \vdash \lambda[X <: \top]t : (\forall[X <: \top]T) \,\hat{}\, C} \ (\textsc{tabs}) \qquad \frac{C; \Gamma \vdash x : (\forall[X <: \top]T) \,\hat{}\, C' \qquad \textbf{cap} \notin \mathrm{dcs}(\Gamma, S)}{C; \Gamma \vdash x[S] : [X := S]T} \ (\textsc{tapp})$$

$$\frac{C; \Gamma \vdash t : T \qquad C; (\Gamma, x : T) \vdash u : U \qquad \Gamma \vdash C, U \text{ wf}}{C; \Gamma \vdash \text{let } x = t \text{ in } u : U} \ (\textsc{let})$$

Subcapturing $\Gamma \vdash C_1 <: C_2$ same as Figure 2 but without the (sc-bound) rule

Subtyping $\Gamma \vdash T_1 <: T_2$

The (top) and (capt) rules are the same as in Figure 2. The (trans) and (refl) rules are the same as in Figure 2 but work on capturing types.

$$\frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash \Box T_1 <: \Box T_2} \ (\textsc{boxed}) \qquad \frac{(\Gamma, c) \vdash T_1 <: T_2}{\Gamma \vdash \forall[c]T_1 <: \forall[c]T_2} \ (\textsc{cfun})$$

$$\frac{\Gamma, x : T_2 \vdash U_1 <: U_2 \qquad \Gamma \vdash T_2 <: T_1 \qquad \alpha_1 \preceq \alpha_2}{\Gamma \vdash \forall^{\alpha_1}(x : T_1)U_1 <: \forall^{\alpha_2}(x : T_2)U_2} \ (\textsc{fun}) \qquad \frac{\Gamma, X <: \top \vdash T_1 <: T_2}{\Gamma \vdash \forall[X <: \top]T_1 <: \forall[X <: \top]T_2} \ (\textsc{tfun})$$

$$\frac{\kappa = (X_1^{\nu_1}, \cdots, X_i^+, \cdots, X_n^{\nu_n}) \mapsto S \in \Theta \qquad \Gamma \vdash T_i <: T_i'}{\Gamma \vdash \kappa[T_1, \cdots, T_i, \cdots, T_n] <: \kappa[T_1, \cdots, T_i', \cdots, T_n]} \ (\textsc{applied-p})$$

$$\frac{\kappa = (X_1^{\nu_1}, \cdots, X_i^-, \cdots, X_n^{\nu_n}) \mapsto S \in \Theta \qquad \Gamma \vdash T_i' <: T_i}{\Gamma \vdash \kappa[T_1, \cdots, T_i, \cdots, T_n] <: \kappa[T_1, \cdots, T_i', \cdots, T_n]} \ (\textsc{applied-m})$$

$$\frac{\kappa = (X_1^{\nu_1}, \cdots, X_n^{\nu_n}) \mapsto S \in \Theta \qquad \forall X_i^+, \textbf{cap} \notin \mathrm{dcs}(\Gamma, T_i)}{\Gamma \vdash \kappa[T_1, \cdots, T_n] <: [X_1 := T_1, \cdots, X_n := T_n]S \qquad \Gamma \vdash [X_1 := T_1, \cdots, X_n := T_n]S <: \kappa[T_1, \cdots, T_n]} \ (\textsc{dealias})$$

Reach Refinement $C \vdash T \rightsquigarrow U$

$$C \vdash \top \rightsquigarrow \top \ (\textsc{r-top}) \qquad C \vdash X \rightsquigarrow X \ (\textsc{r-tvar})$$

$$\frac{D \vdash S \rightsquigarrow S'}{D \vdash S \,\hat{}\, C \rightsquigarrow S' \,\hat{}\, [\textbf{cap} := D]C} \ (\textsc{r-capt}) \qquad \frac{D \vdash T \rightsquigarrow T'}{D \vdash \forall[c]T \rightsquigarrow \forall[c]T'} \ (\textsc{r-cfun}) \qquad \frac{D \vdash T \rightsquigarrow T'}{D \vdash \Box T \rightsquigarrow \Box T'} \ (\textsc{r-boxed})$$

$$\frac{D \vdash T \rightsquigarrow T'}{D \vdash \forall[X <: \top]T \rightsquigarrow \forall[X <: \top]T'} \ (\textsc{r-tfun}) \qquad D \vdash \forall^\alpha(z : T)U \rightsquigarrow \forall^\alpha(z : T)U \ (\textsc{r-fun})$$

$$\frac{\kappa = (X_1^{\nu_1}, \cdots, X_n^{\nu_n}) \mapsto S \in \Theta \qquad \forall X_i^+, D \vdash T_i \rightsquigarrow T_i' \qquad \forall X_i^-, T_i' = T_i}{D \vdash \kappa[T_1, \cdots, T_n] \rightsquigarrow \kappa[T_1', \cdots, T_n']} \ (\textsc{r-applied})$$

Fig. 4. Static semantics of System Reacap.

and (applied-m) supports argument subtyping for applied types with respect to the variance. (dealias) deals with the expansion of applied types.

4.2.3 *Reach Refinement.* Reach refinement $C \vdash T \rightsquigarrow U$ replaces certain covariant occurrences of **cap** in type T with the capture set C, where C is often the reach capability of a variable (e.g., in the (var) rule). As shown in Figure 4, refinement is defined recursively on the structure of types.

The base cases for top types and type variables (R-TOP) and (R-TVAR) leave the type unchanged. For capturing types (R-CAPT), refinement recursively applies to the shape type and directly substitutes occurrences of **cap** in the capture set with C. For boxed types (R-BOXED) and type functions (R-TFUN), refinement is applied to the inner type. For applied types (R-APPLIED), refinement is applied to covariant type arguments and contravariant ones are left unchanged. Reach refinement touches neither the domain nor the codomain of function types (R-FUN), which is due to the translation scheme discussed in Section 2.2. This is further discussed in the technical report [55, Appendix B.3].

4.2.4 Type Definitions. Admittedly, this restriction that prevents refining function types indeed imposes a loss in expressiveness. For example, church-encoded data types cannot be properly refined. Nevertheless, we can recover the expressiveness thanks to type definitions and applied types. As an example, consider the church-encoded pair type:

$$\mathsf{Pair} = (X_1^+, X_2^+) \mapsto \forall[X_R <: \top]\forall(z : X_1 \Rightarrow X_2 \Rightarrow X_R)X_R$$

Assume that the applied type $\mathsf{Pair}[\Box\,\mathsf{IO}\,\hat{}\,\{\mathbf{cap}\}, \Box\,\mathsf{IO}\,\hat{}\,\{\mathbf{cap}\}]$ is bound to a variable x. (R-APPLIED) refines this type to $\mathsf{Pair}[\Box\,\mathsf{IO}\,\hat{}\,\{x^*\}, \Box\,\mathsf{IO}\,\hat{}\,\{x^*\}]$, which can then be expanded to $\forall[X_R <: \top]\forall(z : \Box\,\mathsf{IO}\,\hat{}\,\{x^*\} \Rightarrow \Box\,\mathsf{IO}\,\hat{}\,\{x^*\} \Rightarrow X_R)X_R$ by the (DEALIAS) rule in subtyping. Essentially, applied types change the way the existential quantifications are scoped: In particular, the applied type $\mathsf{Pair}[\Box\,\mathsf{IO}\,\hat{}\,\{\mathbf{cap}\}, \Box\,\mathsf{IO}\,\hat{}\,\{\mathbf{cap}\}]$ is interpreted as $\exists c.\,\mathsf{Pair}[\Box\,\mathsf{IO}\,\hat{}\,\{c\}, \Box\,\mathsf{IO}\,\hat{}\,\{c\}]$. The existential variable is quantified at the outer-level. The **cap**s can thus safely be replaced by x^* in the refinement.

4.3 Translation to System Capless

System Reacap is a surface language that "desugars" to Capless in terms of a type-preserving translation (Section 5.3). The core idea of the translation is to recursively convert the occurrences of **cap** in parameter positions to universal capture parameters, and those in function result types to existential quantifications. Consider the System Reacap type of the mkIterator function (Section 2.2):

$$\forall[X <: \top]\forall^\bullet(x : \mathsf{List}[\Box\,(\mathsf{Unit} \Rightarrow X)])\mathsf{Iterator}[X]\,\hat{}\,\{x^*\}$$

It translates to:

$$\forall[X <: \top]\forall[c_x <: \{\}]\forall[c_{x^*}](\forall(x : \mathsf{List}[\Box\,(\mathsf{Unit} \rightarrow X)\,\hat{}\,\{c_{x^*}\}]\,\hat{}\,\{c_x\})\mathsf{Iterator}[X]\,\hat{}\,\{c_{x^*}\})\,\hat{}\,\{c_{x^*}\}$$

Here, two universal capture parameters c_x and c_{x^*} are introduced. c_x corresponds to the outermost capture set of the parameter x, and is upper-bounded by an empty set in this case. c_{x^*} corresponds to the **cap**s inside the box of x, which is exactly the *meaning* of the reach capability x^*. x is a use-parameter, thus c_{x^*} is allowed to be used in the body of the function.

One surprising aspect of the translation is that subtyping in the source language induces term transformations in the target language. Consider a Reacap term a whose type T is a subtype of U. Let T' and U' be the translated types of T and U respectively, and a' be the translated term of a of type T'. In general, a' does not directly conform to the type U'. It needs to be transformed to a term a'' to conform to the type U'. In fact, widening a capture set C to $\{\mathbf{cap}\}$ in the source language corresponds to packing the capture set into an existential in the target language. For instance, consider the types $\forall(x : \mathsf{File}\,\hat{}\,\{\mathbf{cap}\})\mathsf{File}\,\hat{}\,\{x\}$ and $\forall(x : \mathsf{File}\,\hat{}\,\{\mathbf{cap}\})\mathsf{File}\,\hat{}\,\{\mathbf{cap}\}$. The first type is a subtype of the second in Reacap. However, their translations differ:

$$\forall(x : \mathsf{File}\,\hat{}\,\{\mathbf{cap}\})\mathsf{File}\,\hat{}\,\{x\} \quad \text{translates to} \quad \forall[c_x]\forall(x : \mathsf{File}\,\hat{}\,\{c_x\})\mathsf{File}\,\hat{}\,\{c_x\}$$
$$\forall(x : \mathsf{File}\,\hat{}\,\{\mathbf{cap}\})\mathsf{File}\,\hat{}\,\{\mathbf{cap}\} \quad \text{translates to} \quad \forall[c_x]\forall(x : \mathsf{File}\,\hat{}\,\{c_x\})\exists c.\,\mathsf{File}\,\hat{}\,\{c\}$$

To transform a term of the first translated type to one of the second type, we must perform an eta-expansion to pack the capture set $\{c_x\}$ into the existential quantification:

$$\lambda[c_x]\lambda(x : \mathsf{File}\,\hat{}\,\{c_x\}).\langle\{c_x\}, a_f\,x\rangle$$

where a_f is the translated version of the original function.

The necessity of term transformations motivates us to restrict subtyping on answers (SUB), which ease the transformation process; and to restrict type parameters to be unbounded, due to the lack of

Syntax

$$s,\ t,\ u \coloneqq \mathsf{boundary}[S]\ \mathsf{as}\ \langle c, x\rangle\ \mathsf{in}\ t \mid \cdots \qquad \textbf{Term}$$

$$R,\ S \coloneqq \mathsf{Break}[S] \mid \cdots \qquad \textbf{Shape Type}$$

Subtyping $\Gamma \vdash S_1 <: S_2$

$$\frac{\Gamma \vdash S_2 <: S_1}{\Gamma \vdash \mathsf{Break}[S_1] <: \mathsf{Break}[S_2]}\ (\textsc{break})$$

Typing $\Gamma \vdash t : T$

$$\frac{C; (\Gamma, c : \mathsf{CapSet}, x : \mathsf{Break}[S]\,\hat{}\,\{c\}) \vdash t : S \qquad \Gamma \vdash S\ \mathsf{wf}}{(C \setminus \{c, x\}); \Gamma \vdash \mathsf{boundary}[S]\ \mathsf{as}\ \langle c, x\rangle\ \mathsf{in}\ t : S}\ (\textsc{boundary})$$

$$\frac{C'; \Gamma \vdash x : \mathsf{Break}[S]\,\hat{}\,C \qquad C'; \Gamma \vdash y : S}{C'; \Gamma \vdash x\ y : E}\ (\textsc{invoke})$$

Fig. 5. Extensions to static rules of Capless.

term-level witnesses for subtyping over type parameters. When translating subtyping derivations to System Capless, we require term-level witnesses that can be transformed to adapt between different translated types. However, subtyping between bounded type parameters exists purely at the type level with no corresponding term representation that could be transformed during translation. For instance, consider a type function $f : \forall[X <: (\mathsf{IO}\,\hat{}\,\{\mathsf{io}\} \to \mathsf{Unit})] \ldots$ and an application $f[\mathsf{IO}\,\hat{}\,\{\mathbf{cap}\} \to \mathsf{Unit}]$, where $\mathsf{IO}\,\hat{}\,\{\mathbf{cap}\} \to \mathsf{Unit} <: \mathsf{IO}\,\hat{}\,\{\mathsf{io}\} \to \mathsf{Unit}$. Translating this application form requires the source-language subtyping relation which materializes into term transformation in the target language, but no term is available to be adapted at the type-application site. Since we cannot produce the necessary term-level adaptations for bounded type parameters, we restrict all type parameters to be unbounded, so that the translation remains complete.

5 Metatheory

We prove the type soundness of System Capless (Section 3), through standard progress and preservation theorems as well as its scope safey. Those proofs are fully mechanized in Lean 4.

In addition, we relate the surface-language System Reacap (Section 4) to the core language System Capless via a type-preserving translation. Specifically, any well-typed program in Reacap can be translated to a well-typed one in Capless with an equivalent type.

5.1 Type Soundness

We take a standard syntactic approach towards type soundness, proving the following theorems.

Theorem 5.1 (Preservation). *If (1) $\vdash \Sigma :: \Gamma$, (2) $C; \Gamma \vdash t : E$, and (3) $\langle \Sigma \mid t\rangle \longrightarrow \langle \Sigma' \mid t'\rangle$, then there exist C' and Δ such that (1) $\vdash \Sigma' :: (\Gamma, \Delta)$; and (2) $C'; (\Gamma, \Delta) \vdash t' : E$.*

Here, $\vdash \Sigma :: \Gamma$ denotes *store typing*: given a store Σ, this relation produces a typing context Γ whose bindings assign a type to each corresponding value in the store (defined in the technical report [55, Figure 11]). $\langle \Sigma \mid t\rangle \longrightarrow \langle \Sigma' \mid t'\rangle$ denotes the reduction relation.

Theorem 5.2 (Progress). *Given (1) $\vdash \Sigma :: \Gamma$, and (2) $C; \Gamma \vdash t : E$, we can show that either t is an answer, or there exist Σ' and t' such that $\langle \Sigma \mid t\rangle \longrightarrow \langle \Sigma' \mid t'\rangle$.*

5.2 Scope Safety

Following Boruch-Gruszecki et al., we add an extension to System Capless which introduces scoped capabilities. By proving it sound, we show that the system can ensure the scoping of capabilities.

5.2.1 Static Semantics of Scoped Capabilities. Figure 5 presents the extensions. We add a control delimiter $\mathsf{boundary}[S]\ \mathsf{as}\ \langle c, x\rangle\ \mathsf{in}\ t$ (mirroring `boundary/Break` in Scala 3) that introduces a scope with a fresh abstract capture c and a scoped capability $\mathsf{Break}[S]$. The capability may be used only within its defining boundary; the type system enforces this as follows: (BOUNDARY) checks the body under Break, binds the fresh c, and requires the result type S to be well-formed in the outer context Γ, which suffices to enforce scoping. (INVOKE) types invocations of Break, and (BREAK) provides subtyping between Break capabilities.

5.2.2 Dynamic Semantics of Scoped Capabilities. In the reduction semantics we introduce (1) runtime labels to identify boundaries and (2) a scoping construct that delimits a boundary together with its Break capability. Invoking Break requires a matching scope in the surrounding evaluation context; otherwise evaluation is stuck. For brevity, we elide the full runtime forms here and refer to the technical report [55, Appendix A.3] for the complete definition.

5.2.3 Type Soundness. We prove standard progress and preservation theorems to establish the type soundness of the extended system. Its type soundness implies that the system enforces the scoping discipline of the Break capability, as a scope extrusion will lead to a stuck evaluation. The proof is fully mechanized in Lean 4.

5.3 Type Preserving Translation

We develop a translation system that translates well-typed programs in System Reacap to System Capless. This essentially provides an understanding of **cap**s and reach capabilities: they can be understood as existential and universal quantifications of capture sets. The translation is stated and proven with pen and paper. See our technical report [55, Appendix D] for the full details.

Theorem 5.3 (Translation Preserves Typing). *Let $\tau = \langle D, \rho, \rho^* \rangle$ be a proper translation context under type contexts Γ and Δ, and $C; \Gamma \vdash t : T$ be a typing derivation in System Reacap. Then, if t is either of the application form $x\,y$ or the let-binding form let $z = s$ in u, there exists a term t' in System Capless such that $[\![C]\!]^{D'}; \Delta \vdash t' : \exists c.\, [\![T]\!]^{\{c\}}$ for some D'; otherwise, there exists a term t' in System Capless such that $[\![C]\!]^{D'}; \Delta \vdash t' : [\![T]\!]^{D'}$ for some D', and t' is an answer when t is an answer.*

$\langle D, \rho, \rho^* \rangle$ is the translation context in System Decap, a translation system that converts capture sets and types in Reacap into those in Capless. The translation context consists of (1) a capture set D assigning meaning to **cap**s, (2) and two mappings ρ and ρ^* that map capabilities (including reach capabilities) to their corresponding capture sets. The notation $[\![T]\!]^D$ denotes the translation of a Reacap type T to a Capless type, and similarly for $[\![C]\!]^D$, which denotes the translation of a Reacap capture set C. The key idea of the translation is to interpret **cap**s in the argument types as universal quantifications and those occuring covariantly in the result type as existential quantifications.

6 Capture Tracking for Asynchronous Programming: A Case Study

We study the usage of capture checking and more specifically reach capabilities through some real-world examples of asynchronous programming.

Futures and Scoping. Scala supports a simple form of concurrency with Futures. A `Future` takes a computation and runs it concurrently under a provided execution context. Note that despite the computation not being stored by the Future, it can hold on to capabilities captured by the computation after `Future.apply` returns. Therefore, a Future should capture both the execution context and its computation. We model the Future constructor using a context function [37]:

```
object Future { def apply[T](comp: => T)(using ec: ExecutionContext^): Future[T]^{ec, comp} }
```

The following program attempts to read a file and filter its content, using the standard `Using` resource-pattern. However, notice that `Future.apply` does not run the reading procedure to the end, instead returning immediately and causing `Using` to close the file right after.

```
def findLines(p: String)(using ec: ExecutionContext^): Future[Seq[String]]^{ec} =
  Using(File.open("text.txt")): (file: File^) => // ERROR: file leaked
```

```
    Future.apply:                 // : Future[Seq[String]]^{ec, file}
      file.readLines()            // : Iterator[String]^{file}
          .filter(_.contains(p)) // : Iterator[String]^{file}
          .toSeq                  // : Seq[String]
```

The compiler rejects it, detecting that `Future.apply` returns a `Future` capturing `file` which escapes the `Using` scope. To fix this, the file has to be opened *under* the `Future.apply` call.

Composing Futures. When writing concurrency code, it is common to create multiple concurrent computations, and then combine them either by requiring both or either (racing) futures:

```
extension [T](@use xs: Seq[Future[T]^])
  def all(using ec: ExecutionContext^): Future[Seq[T]]^{ec, xs*}
  def first(using ec: ExecutionContext^): Future[T]^{ec, xs*}
```

The extension methods `all` and `first` create a future that combines or races the input futures. The resulting `Future` upholds all scoping restrictions of the futures in the input sequence by capturing the reach capability of the sequence itself. As we saw in Section 2.2, `xs` is pure, but we want to "open" the sequence of futures to await for their results, effectively using the futures' captures. For this reason, the capture checker requires the **`@use`** annotation; and when added, it will propagate the reach capability `xs*` to the capture set of the caller. This is demonstrated in the following example:

```
withIO:
  withThrow:
    val futs: Seq[Future[Int]^{async, io, throw}] =
      val f1: Future[Int]^{async, io} = Future(useIO())
      val f2: Future[Int]^{async, throw} = Future(useThrow())
    () => futs.all // (() -> Future[Seq[Int]])^{async, io, throw}
```

The returned function only captures `futs`, a pure variable. However, **`@use`** requires the `.all` call to add the reach capabilities of `futs` to the closure, resulting in the rejection of the program. Without this check, we would be able to invoke `io` and **`throw`** outside of their scope.

The implementation of `.all` uses a similar construct to the `collect` example in Section 1, named `Future.Collector[T]`. A collector takes a sequence `xs` of `Future[T]` and exposes a read-only channel that passes back the futures as they are completed. Naturally, the returned futures have the same capture set as the union of all captures of the input futures – or the reach capability `xs*`. To implement `.all`, we create a `Collector` from the given futures, and chain the results on each as they are completed - guaranteeing that failing futures immediately return the exception to the caller as soon as possible:

```
extension [T](@use xs: Seq[Future[T]^]) def all(using ExecutionContext^) =
  val results = Collector(xs).results // : Channel[Future[T]^{xs*}]
  // create a future that poll the results channel xs.size times
  val poll = (0 until xs.size)
    .foldLeft(Future.unit)((fut, _) => fut.andThen(_ => results.read()))
  poll.andThen: _ => // all futures are resolved at this point
    xs.foldLeft(Future.apply(Seq.empty)): (seq, fut) =>
        seq.zip(fut).map(_ +: _) // : Future[Seq[T]]^{xs*}
```

Mutable Collectors. A *mutable* version of `Collector[T]` is useful for work queues where jobs dynamically spawn new concurrent tasks, but results should arrive as soon as possible. Unfortunately, for mutable collections like `MutableCollector[T]`, reach capabilities are not expressive enough: the collection may be created empty (and hence no longer accept inserting capturing values), but allowing the capture set of the collection's items to grow is unsound. In such cases, we reach a middle ground and require the user to *declare in advance* the upper-bound capture set, with an explicit parameter. The collection then can accept any item conforming to this declared capture set:

```
class MutableCollector[T, C^]():
  val results: ReadChannel[Future[T]^{C}]; def remaining: Int
  def add(fut: Future[T]^{C})
def parSearch(run: Node => Seq[Node], start: Node)(using ec: ExecutionContext^) =
  val queue = MutableCollector[List[Node], {ec, run}]()
  queue += Future.apply(Seq(start))
  def loop(): Future[Unit]^{ec, run} =
    if queue.remaining == 0 then Future.unit else // : Future[Unit]^{}
      val nextToProcess = queue.results.read() // : Future[Seq[Node]]^{ec, run}
      nextToProcess.andThen: children =>
        children.foreach(node => queue += Future.apply(run(node)))
        loop()
  loop()
```

The above example illustrates the use of mutable collectors to implement parallel tree search. In `parSearch`, we create a mutable collector that accepts `Futures` that can capture the execution context and capabilities used by `run`. We utilize the collector as a work queue to immediately spawn new jobs as soon as one comes back, resulting in minimal downtime waiting while more children are discovered. To specify a capture set, we use the `{ }` syntax in the type parameter. Note that there is very little notational overhead when implementing `parSearch`: the mutable collector requires a non-inferrable capture set, and `loop`, being recursive, requires a specified return type. All other type annotations in comments, including capture annotations, are inferred by the compiler.

7 Evaluation

While the previous case study demonstrated the practicality of our approach through qualitative analysis of common programming patterns, this section provides a quantitative evaluation by measuring performance and required code changes when adopting capture checking in practice.

7.1 Implementation

The Scala 3 capture checker provides a complete implementation of capture tracking with reach capabilities, as well as optional universal quantification of capture sets. It is enabled by an experimental language import.

Table 1. Compiler performance compiling the capture-checked standard library. Average of 10 runs, measured on Linux PC (Ryzen 3800x, 32GB RAM, NixOS, JDK11)

Phase	Time (ms)	Throughput
Typing	10436 ± 220 (35.80%)	4596 ± 99 loc/s
CC	3896 ± 48 (13.36%)	12307 ± 148 loc/s
Other	14821 ± 269 (50.84%)	3236 ± 58 loc/s
Total	29154 ± 395 (100.0%)	1645 ± 22 loc/s

The capture checker runs after the type checker and several transformation phases. It re-checks the typed syntax tree of a compilation unit, deriving subcapturing constraints between capture sets that are solved incrementally with a propagation-based constraint solver that keeps track of which capabilities are known to form part of a capture set. To understand the performance impact of capture checking, we show the compilation time breakdown of compiling the capture-checked standard library in Table 1. Capture checking takes approximately half the time compared to the previous typing phase and accounts for less than 15% of the total compilation time, which is reasonable. All type arguments are treated as boxed, and boxing and unboxing operations are inferred in an adaptation step that compares actual against expected types.

7.2 Porting the Scala Standard Library

To evaluate the practicality of capture checking, we have ported a significant portion of the Scala standard library to compile with the Scala 3 capture checker. In particular, the collections library is ported in full, alongside common language structures like `Array` and `Function` traits. This shall be referred to as `lib-cc`, as opposed to the original standard library (`lib`), which is at version 2.13.15 at the time of comparison.

We measure the number of changes by direct comparison (using standard `diff`) between `lib-cc` and `lib` source code. The changes are categorized into Table 2. Overall, capture checking annotations require only about 3% of lines of code to be changed, with over half of the changes involving only adding capture sets to function and class declarations. Despite the vast majority (>98.5%) of `lib-cc` being generic collections, about 87% of all classes and 88% of methods do not require any signature change - all existing code will compile as-is. This is thanks to boxing type parameters as well as built-in type and capture set inference.

Table 2. Number of changes to capture-checked standard library, grouped by categories of change. "Total in lib-cc" shows the total number of definitions, variables, higher order functions, casts and lines of code in the library. "Meaningful" measures lines of changes excluding whole-line comments, blank lines and feature imports. For reference, "Total in lib" shows corresponding numbers in the original standard library (Scala 2.13.15), including parts not yet implemented by lib-cc, i.e., 66% of the standard library is collections.

Type of Change	**Added/Changed**	**Total in lib-cc**	***Total in lib***
Capture sets on definitions	691 functions 91 classes	6189 functions 684 classes	*11113 functions* *1518 classes*
- Only universal captures	351		
- Only capture set on returns	379		
Capture sets on local variables	53	4583	*6262*
Restrict functions to pure	52	797	*1203*
Reach capabilities	19		
@use annotations	12		
Unsafe casts	5	810	*1325*
Unsafe capture set removal	25		
Class hierarchy changes	2		
- New classes	3		
Total lines	+1418/-1407 meaningful	52160 total 31395 code	*94635 total* *48210 code*

7.2.1 Additional Annotations. In a lot of cases, the existing signatures work as it is. Examples are higher order functions like `List.map(f: A => B)`, whose parameters are only used and not captured in the return value. Scala's capture checking implementation treats "fat-arrow" function types as effect-polymorphic, and therefore `List.map` does not require any changes. There are some false-positives, where a pure function is expected: the `maxBy` method on an `IterableOnce[A]` expects a function that extracts a property of `A`, and should not perform any effects. We restrict functions to pure (`A -> B`) in such cases. For many other cases, the existing implementation is sufficient to handle arbitrarily capturing inputs and outputs, but extra capture set annotations are needed to communicate this fact. One of such cases involves passing parameters of a possibly capturing type, such as an iterator in the following method of `List[A]`:

```
def prependAll(it: IterableOnce[A]^): List[A]
```

In this case, `prependAll` should consume items from the given iterator (possibly invoking capabilities captured in it), and then return a new prepended list without capturing `it`. As-is, `prependAll` would only allow pure iterators to be passed in, greatly reducing its usefulness. However, adding the universal capture annotation completely fixes this issue. Over half (51.7%) of all changed signatures involve only this addition of **`cap`**.

In some cases, the return value captures one or more of its parameters (and possibly the current **`this`**). Of particular note is the implementation of functional operations of iterators and other lazy data structures, e.g., the following signature of `map` on an `Iterator[A]`:

```
def map[B](f: A => B): Iterator[B]^{this, f}
```

Note that the return type captures both the original iterator (**`this`**) and the mapping function; and this is also the only form of additional annotation required. Such changes are very common, and make up of most of the remaining half of changes on signatures.

As expected, all instances of reach captures belong to `flatMap` (and `flatten`, which is a special case of `flatMap`) implementations of (possibly-)lazy data structures and interfaces. All such instances in parameter position show up with `@use` annotations. Both can be removed when overriding such interfaces in a strict data structure like `Map`.

7.2.2 Guaranteeing Compatibility. To enable incremental adoption, Scala allows mixing capture-checked and unchecked modules. However, compiling them together still presents challenges. One current limitation is that explicit capture-set parameters cannot yet be used from unchecked modules. For example, the class `ConcatIterator[A]` internally maintains a mutable linked list:

```
class ConcatIterator[A](var iterators: mutable.List[IterableOnce[A]^]):
  // concatenate `it` with `this` into a new iterator
  def concat(it: IterableOnce[A]^): ConcatIterator[A]^{this, it} = iterators ++= it.unsafeAssumePure
```

Similar to the mutable collectors in Section 6, such lists require an explicit capture parameter to track its elements' capture sets. To preserve compatibility, we instead opt for external tracking: moving the capture set of the list to the `ConcatIterator` itself. This is unsafe (`concat` returns a new iterator reference with expanded captures, but the old reference is still valid). Future work aims to lift this restriction safely.

Furthermore, interoperability between modules using `lib-cc` and `lib` necessitates strict binary compatibility: no new classes or hierarchy changes. Fortunately, most collection classes are individually capture-checked without such changes. One notable example is `IndexedSeqView[A]` - a lazy view on a sequential collection - unsoundly implementing non-capturing operations similar to `List.map`. To resolve this, we introduced a separate class hierarchy, achieving capture soundness at the cost of potential link-time errors, which are easily identified during development.

8 Discussion

Towards fully path-dependent capabilities. Reach capabilities alone are coarse-grained, because they track all the capabilities of a whole data structure rather than the capabilities of individual elements. E.g., they do not reflect that a function only accesses the first element of a pair:

```
def fst(@use p: (() => Unit, () => Unit)): () ->{p*} Unit = p._1
val p: (() ->{io} Unit, () ->{fs} Unit) = ...
fst(p) // : () ->{io, fs} Unit instead of () ->{io} Unit
```

Passing `p` to `takeFirst` yields a result of type `() ->{io, fs} Unit`, capturing capabilities from both components even though only the first component is returned. As a workaround, a precise version can be defined in our system by falling back to explicit capture polymorphism:

```
def fstAlt[C1, C2](p: (() ->{C1} Unit, () ->{C2} Unit)): () ->{C1} Unit
```

This approach, however, is antithetical to the lightweight nature of our system, which aims to avoid explicit polymorphism in most cases. Has the lightweight syntax run out of steam already?

Fortunately, the expressive power of reach capabilities and the lightweight syntax can be greatly amplified through Scala's fully *path-dependent types*, generalizing reach capabilities on one variable to full paths. We can give precise types to functions like `fst` and `snd` without resorting to explicit capture polymorphism by capturing the paths to the individual components:

```
def fst( @use p: (() => Unit, () => Unit)): () ->{p._1*} Unit = p._1
def snd( @use p: (() => Unit, () => Unit)): () ->{p._2*} Unit = p._2
def copy(@use p: (() => Unit, () => Unit)): (() ->{p._1*} Unit, () ->{p._2*} Unit) = (fst(p),snd(p))
```

Indeed, the Scala capture checker already supports full paths in this way.

Scala also supports "capture-set members," enabling capture polymorphism like in DOT [45]:

```
trait CaptureSet { type C^ /* declare capture-set member */ }
def capturePoly(c: CaptureSet)(f: File^{c.C}): File^{c.C} = f
```

We leave the exploration of the underlying theory for future work.

Telling a Use from a Mention. Another limitation of our system is that it cannot precisely distinguish a "mention" of a variable from an actual "use" of it [19]. This stems from the (VAR) rule in Figure 2, which always adds referenced variables to the use set. For example:

```
val f: () => Unit = ...
val g = () => f // : () ->{f} () ->{f} Unit
```

Although g does not invoke f and only returns it, f appears in its capture set, making g impure. However, an equivalent yet pure version can be defined with explicit eta-expansion:

```
val g1 = () => () => f() // : () -> () ->{f} Unit
```

This alternative specifies the evaluation order more precisely, allowing the outer function to be pure while the inner function captures f.

Our evaluation (Section 7) suggests this limitation is benign in practice. We encountered no blockers while porting the Scala standard library. Accurately tracking such fine-grained patterns would require a substantially richer system and remains interesting future work.

Fresh Out of the Box: Reasoning about Aliasing and Separation. Capture checking already shows promise toward fearless concurrency [53] and a solution to the "what-color-is-your-function?" problem [36] for Scala 3. To be fully effective in these areas, we plan to add reasoning about aliasing, separation, and capability exclusivity, leading to a system that can express Rust-style ownership patterns and advanced uses like safe manual memory management.

Reachability types (RT) [3, 51] (an independent line of work, unrelated to the Scala 3 capture-checking effort) demonstrate that capture tracking supports reasoning about aliasing and separation, powering compiler optimizations for impure functional DSLs in the LMS library for Scala 2 [8, 7].

RT is a box-free system[4] that handles generics by threading explicit type-and-capture quantifiers through every definition [51]. It therefore faces the same dilemma as other box-free polymorphic systems of requiring invasive changes (cf. Section 9). Our work for Scala 3 retrofits the existing ecosystem as much as possible via boxing, perhaps the only realistic option to introduce CT into Scala without a complete overhaul. That would alienate existing users.

Furthermore, Wei et al. [51] point out an issue of CT's boxing mechanism and the way it is used for escape-checking (see Section 2.1.3) in conjunction with freshly allocated objects:

```
withFile("file.txt"): f =>
  val r = new Ref(0) // Ref[Int]^, a freshly allocated object, would be Ref[Int]♦ in RT
  r                  // error, cannot unbox Ref[Int]^ in CT, but legal in RT
```

The most natural capture set for fresh values in CT is the top capability **cap**, but then it is impossible for such values to escape the scope they were created in. Hence, Wei et al. reject having **cap** and instead assign a "freshness marker" ♦ to fresh values, which is not a subcapturing top element.

System Capless (Section 3) replaces the top capability **cap** with explicit quantification under the hood. This yields a principled basis for freshness: whereas **cap** denotes "arbitrary unknown capabilities," we can extend existential quantifiers to distinguish *arbitrary* from *fresh* capabilities.

For instance, with such extensions, the function mkRef that creates a fresh Ref[Int] can be typed as Int -> $\exists c : \mathsf{Fresh}.$ Ref[Int]^{c}, where $\exists c : \mathsf{Fresh}. T$ represents a fresh existential type. In the surface language, the type can be simply Int -> Ref[Int]^, with the hat ^ denoting an existential. This would enable the following example:

```
def makeCounter(init: Int): ∃c : Fresh. Pair[box () ->{c} Int, box () ->{c} Int] =
  val r = mkRef(init)
  Pair(box(() => r.get), box(() => r.set(r.get + 1)))
val counter = makeCounter(0) // c: Fresh, counter: Pair[box () ->{c} Int, box () ->{c} Int]
val get = unbox(counter.snd) // : () ->{counter} Int // ok
```

[4]Also note that our reach capabilities have no analogue in RT due to the absence of boxes.

The last line is legal because `c` is *fresh*, which means that it represents capabilities that are guaranteed to not conflict with any existing scope, making unboxing safe without escape concerns. This example is also supported by RT through their self-references [51], which allow function types to refer to their own capture sets. While both approaches express this pattern, existential quantification arguably provides a more natural and principled representation.

To summarize, CT embed capturing types via boxing, whereas RT rely on explicit type-and-capture parameters threaded through every generic definition, and a different subcapturing relation. The approaches are therefore already orthogonal, and the gap will widen as we progress on adding a separation-checking layer on top of System Capless in the future.

Categorizing Capabilities. Our system covers Scala's standard *collections* library, but not yet the *full* standard library. A notable gap is `Try[T]`, a sum of a value `T` and a (pure) exception. Its constructor accepts a by-name `T`, evaluates it, and captures any thrown value; clients later call `.get` to obtain `T` or rethrow. With the current system we type:

```
def apply[T](body: => T): Try[T] // should be Try[T]^{body.only[Control]}
```

Here `body` is (unboundedly) capture-polymorphic while `Try[T]` is pure. Although that matches the value-level view (a boxed `T` or a pure exception), it breaks the exception discipline enforced by Scala's `CanThrow` capabilities [11, 38]: `.get` can rethrow without the corresponding capability. From the capability view, the constructor should instead record a *subset* of `body`'s captures related to exception-like control transfers: a class we call *Control Capabilities*. Expressing this requires extending CC with (1) *capability categories*, and (2) *category-restricted subsets* (e.g. `body.only[Control]`).

Such categories arise elsewhere: label-safe delimited continuations require restricting captured labels [40]; thread spawning must forbid capturing thread-local resources (e.g. mutex guards, stack pointers, non-atomic counters); and mutability can be treated as its own category. We therefore see capability categorization, especially for asynchronous code, as promising future work.

9 Related Work

To our knowledge, this is the first work that introduces an opt-in type system for tracking effects-as-capabilities into a mainstream programming language, applying it to a widely-used standard collections library. Although prior works have addressed effect tracking, capabilities, or resources, they typically focus on effects or ownership rather than directly capturing retained capabilities. Concerns about retrofitting such systems into existing languages (especially in a non-invasive and pragmatic manner as in our work) are rarely addressed in the literature.

Polymorphism across Capabilities, Effects, and Ownership. Polymorphism is indispensable: abstractions must work no matter which effect, resource, or ownership context they run in. Classic polymorphic effect systems [27, 46] show how to track side effects, while region typing [48, 20], uniqueness and linearity [49, 4, 29], and ownership types [12, 35, 6, 42, 16] control memory and aliasing. Yet all of these scatter extra indices, regions, uniqueness flags, or ownership parameters throughout every signature. Our capture-checking design avoids that clutter: it reuses Scala's path-dependent types for implicit polymorphism, and embeds capture-tracked types through boxing, inspired by CMTT [34] and Effekt [9]. Boxing gently embeds a new type universe into the existing one: existing type variables in old signatures can range over capturing types while preserving parametricity. Consequently, the standard collections API needs no new type parameters, adoption is strictly opt-in, and code that ignores capture checking remains fully source-compatible. Optional explicit capture polymorphism is supported but rarely needed in practice (Section 7.2).

Practical Ownership and Capability Languages. Rust [30] employs an ownership model to ensure memory safety without garbage collection. Its type system infers ownership, borrowing, and lifetimes but imposes restrictive invariants, complicating higher-order functional programming as well as implementations of data structures, like doubly-linked lists. Pony [41, 13, 14] uses object and reference capabilities to control mutability and aliasing, achieving capability polymorphism through complex viewpoint adaptation rules. Mezzo [2] tracks shared, exclusive, and immutable ownership implicitly via permissions, but still requires explicit permission polymorphism for generic signatures. Project Verona [1] simplifies memory management with hierarchical regions and ownership semantics. It manages capabilities via isolated regions and viewpoint adaptation through method overloading.

These systems exemplify qualified type systems [24, 18, 22]. Our work differs by qualifying types explicitly with retained capabilities rather than permissions or regions, aligning more closely with pure object capabilities [33]. The primary novelty compared to previous CT work [5] is introducing reach capabilities, for which we are hard-pressed to find a direct analogue in the literature.

OCaml Modal Types. Efforts to bring Rust-style ownership to OCaml [47, 26] adopt a modal type system that tracks an ambient effect context for enabling stack allocation and preventing data races. Section 8 sketches how Scala's capture checking can offer similar guarantees. Like our work, these approaches aim for backward compatibility and lightweight signatures, yet they rely on Hindley-Milner inference rather than our local type inference with path-dependent types. Tang et al. [47] extend the modal approach to Frank-style effect handlers [25, 15], tracking effects via the modal context rather than capture sets. Both systems sometimes require explicit polymorphism. We look forward to the final implementation and to clarifying the precise connection between modal and capturing types.

Path-dependent Types and Type members for Effects and Capabilities. Wyvern [32, 31, 17] builds on object-capabilities [33], first estimating authority with a whole-program analysis [31] and later adding a path-dependent effect system [32] inspired by DOT [45]. Each object can declare an abstract effect member that upper- or lower-bounds the effects its methods may perform, mirroring DOT's abstract type members. Our capture checking takes the complementary view: we record the objects a value can reach in its capture set; the allowable effects follow implicitly from those objects' APIs. Thus both systems use the same abstraction and composition machinery, but Wyvern names effects directly, while Scala names the resources through which those effects can occur.

Similarly, associated effects for Flix [28] extend the same idea to type classes: each class can declare an abstract effect row, giving first-class, per-instance effect polymorphism without extra type parameters. Like Wyvern, it abstracts over effects; capture sets instead abstract over reachable resources, thereby opening the door to providing alias-control guarantees (cf. Section 8).

Second-Class Values. Osvald et al. [39] introduced 2nd-class values for Scala 2, i.e., values whose lifetime is tied to their lexical scope, but their stratification hinders higher-order programming, restricting currying and lazy collections. Xhebraj et al. [52] extend the idea with 2nd-class returns (recovering currying) and explicit storage-mode polymorphism, yet rely on an unusual stack semantics. Being based on Scala 2 annotations, their implementation cannot express fully storage-mode generic types such as `List[Q, T @mode[Q]]`. Our boxing-based system avoids these limitations. It does not yet support privilege lattices or bounding a closure's free variables, features useful for fractional capabilities [52, Fig. 7], which we expect to model via capability categories (Section 8).

10 Conclusion

This paper introduced reach capabilities and developed their foundations, backed by mechanized type soundness and scope safety proofs. Reach capabilities made it possible to migrate the Scala 3 standard collections library to capture tracking with minimal adjustments. Our approach effectively addresses limitations in handling capabilities within generic data structures and collections, a significant milestone towards bringing lightweight and ergonomic effect systems to the masses.

Acknowledgments

We thank Eugène Flesselle and the anonymous reviewers for their valuable feedback. This work is supported by the SNSF Advanced Grant 209506, "Capabilities for Typing Resources and Effects".

Data-Availability Statement

The artifacts accompanying this paper [54] include: (1) a complete mechanized formalization of System Capless in Lean 4, including the scope safety extension and associated metatheory; (2) a practical implementation of capture checking integrated into the Scala 3 compiler; and (3) source code and scripts for reproducing the paper's case studies. The capture-checked version of Scala's standard collections library is included among these artifacts.

References

[1] Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference capabilities for flexible memory management. *Proc. ACM Program. Lang.*, 7, OOPSLA2, 1363–1393 (cit. on p. 25).

[2] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The design and formalization of Mezzo, a permission-based programming language. *ACM Trans. Program. Lang. Syst.*, 38, 4, 14:1–14:94 (cit. on p. 25).

[3] 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, 1–32 (cit. on p. 23).

[4] Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness typing for functional languages with graph rewriting semantics. *Math. Struct. Comput. Sci.*, 6, 6, 579–612 (cit. on p. 24).

[5] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäuser. 2023. Capturing types. *ACM Trans. Program. Lang. Syst.*, 45, 4, 21:1–21:52 (cit. on pp. 1–5, 7, 10–14, 17, 25).

[6] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. 2003. Ownership types for object encapsulation. In *POPL*. ACM, 213–223 (cit. on p. 24).

[7] Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for impure higher-order languages (technical report). *CoRR*, abs/2309.08118 (cit. on p. 23).

[8] Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for impure higher-order languages: Making aggressive optimizations affordable with precise effect dependencies. *Proc. ACM Program. Lang.*, 7, OOPSLA2, 400–430 (cit. on p. 23).

[9] 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, 1–30 (cit. on pp. 1, 2, 24).

[10] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: Effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4, OOPSLA, 126:1–126:30 (cit. on pp. 1, 2).

[11] 2021. Canthrow capabilities. Accessed: 2025-07-28. https://docs.scala-lang.org/scala3/reference/experimental/canthrow.html (cit. on p. 24).

[12] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership types: A survey. In *Aliasing in Object-Oriented Programming*. Lecture Notes in Computer Science. Vol. 7850. Springer, 15–58 (cit. on p. 24).

[13] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *AGERE!@SPLASH*. ACM, 1–12 (cit. on p. 25).

[14] Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. 2017. Orca: GC and type system co-design for actor languages. *Proc. ACM Program. Lang.*, 1, OOPSLA, 72:1–72:28 (cit. on p. 25).

[15] Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo bee doo bee doo. *J. Funct. Program.*, 30, e9 (cit. on pp. 1, 2, 25).

[16] Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic universe types. In *ECOOP* (Lecture Notes in Computer Science). Vol. 4609. Springer, 28–53 (cit. on p. 24).

[17] Jennifer A. Fish, Darya Melicher, and Jonathan Aldrich. 2020. A case study in language-based security: Building an I/O library for Wyvern. In *Onward!* ACM, 34–47 (cit. on p. 25).

[18] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A theory of type qualifiers. In *PLDI*. ACM, 192–203 (cit. on p. 25).

[19] Colin S. Gordon. 2020. Designing with Static Capabilities and Effects: Use, Mention, and Invariants. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)* (Leibniz International Proceedings in Informatics (LIPIcs)). Robert Hirschfeld and Tobias Pape, (Eds.) Vol. 166. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:25. ISBN: 978-3-95977-154-2. doi:10.4230/LIPIcs.ECOOP.2020.10 (cit. on p. 23).

[20] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-based memory management in Cyclone. In *PLDI*. ACM, 282–293 (cit. on p. 24).

[21] John Hatcliff and Olivier Danvy. 1994. A generic account of continuation-passing styles. In *POPL*. ACM Press, 458–471 (cit. on pp. 11, 13).

[22] Mark P. Jones. 1994. A theory of qualified types. *Sci. Comput. Program.*, 22, 3, 231–256 (cit. on p. 25).

[23] 2015. Leaking local reach capability. Accessed: 2025-03-23. https://github.com/scala/scala3/issues/21442 (cit. on pp. 3, 8).

[24] Edward Lee, Yaoyu Zhao, Ondrej Lhoták, James You, Kavin Satheeskumar, and Jonathan Immanuel Brachthäuser. 2024. Qualifying system $F_{<:}$ Some terms and conditions may apply. *Proc. ACM Program. Lang.*, 8, OOPSLA1, 583–612 (cit. on p. 25).

[25] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *POPL*. ACM, 500–514 (cit. on pp. 1, 2, 25).

[26] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with modal memory management. *Proc. ACM Program. Lang.*, 8, ICFP, Article 253, (Aug. 2024), 30 pages (cit. on p. 25).

[27] John M. Lucassen and David K. Gifford. 1988. Polymorphic effect systems. In *POPL*. ACM Press, 47–57 (cit. on p. 24).

[28] Matthew Lutze and Magnus Madsen. 2024. Associated effects: Flexible abstractions for effectful programming. *Proc. ACM Program. Lang.*, 8, PLDI, 394–416 (cit. on p. 25).

[29] Danielle Marshall, Michael Vollmer, and Dominic Orchard. 2022. Linearity and uniqueness: An entente cordiale. In *ESOP* (Lecture Notes in Computer Science). Vol. 13240. Springer, 346–375 (cit. on p. 24).

[30] Nicholas D. Matsakis and Felix S. Klock II. 2014. The Rust language. In *HILT*. ACM, 103–104 (cit. on p. 24).

[31] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A capability-based module system for authority control. In *ECOOP* (LIPIcs). Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:27 (cit. on p. 25).

[32] Darya Melicher, Anlun Xu, Valerie Zhao, Alex Potanin, and Jonathan Aldrich. 2022. Bounded abstract effects. *ACM Trans. Program. Lang. Syst.*, 44, 1, 5:1–5:48 (cit. on p. 25).

[33] Mark S. Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. Dissertation. John Hopkins University (cit. on pp. 2, 25).

[34] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9, 3, 23:1–23:49 (cit. on p. 24).

[35] James Noble, Jan Vitek, and John Potter. 1998. Flexible alias protection. In *ECOOP* (Lecture Notes in Computer Science). Vol. 1445. Springer, 158–185 (cit. on p. 24).

[36] Robert Nystrom. 2015. What colour is your function? Accessed: 2024-09-09. https://web.archive.org/web/20241009152925/https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/ (cit. on p. 23).

[37] 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, 42:1–42:29 (cit. on p. 18).

[38] Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondrej Lhoták. 2021. Safer exceptions for Scala. In *SCALA/SPLASH*. ACM, 1–11 (cit. on pp. 5, 24).

[39] 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 (cit. on p. 25).

[40] Cao Nguyen Pham and Martin Odersky. 2024. Stack-copying delimited continuations for Scala Native. In *ICOOOLPS @ ECOOP*. ACM, 2–13. doi:10.1145/3679005.3685979 (cit. on p. 24).

[41] [SW Mod.] Pony, Pony Programming Language 2024 Pony Development Team. URL: https://web.archive.org/web/20241007175842/https://www.ponylang.io/, VCS: https://github.com/ponylang/ponyc (cit. on p. 25).

[42] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. 2006. Generic ownership for generic Java. In *OOPSLA*. ACM, 311–324 (cit. on p. 24).

[43] 2015. Reach capabilities get dropped in cv. Accessed: 2025-03-23. https://github.com/scala/scala3/issues/19571 (cit. on pp. 3, 8).

[44] 2015. Reach capabilities of function arguments get ignored. Accessed: 2025-03-23. https://github.com/scala/scala3/issues/20503 (cit. on pp. 3, 8).

[45] Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *OOPSLA*. ACM, 624–641 (cit. on pp. 22, 25).

[46] Jean-Pierre Talpin and Pierre Jouvelot. 1994. The type and effect discipline. *Inf. Comput.*, 111, 2, 245–296 (cit. on p. 24).

[47] Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2025. Modal effect types. *Proc. ACM Program. Lang.*, 9, OOPSLA1, Article 120, (Apr. 2025), 28 pages. doi:10.1145/3720476 (cit. on pp. 1, 2, 25).

[48] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Inf. Comput.*, 132, 2, 109–176 (cit. on p. 24).

[49] Philip Wadler. 1990. Linear types can change the world! In *Programming Concepts and Methods*. North-Holland, 561 (cit. on p. 24).

[50] Philip Wadler. 1989. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (FPCA '89). Association for Computing Machinery, Imperial College, London, United Kingdom, 347–359. ISBN: 0897913280. doi:10.1145/99370.99404 (cit. on p. 7).

[51] Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic reachability types: Tracking freshness, aliasing, and separation in higher-order generic programs. *Proc. ACM Program. Lang.*, 8, POPL, 393–424 (cit. on pp. 23, 24).

[52] Anxhelo Xhebraj, Oliver Bračevac, 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 (cit. on p. 25).

[53] 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, 1181–1207. doi:10.1145/3649853 (cit. on p. 23).

[54] [SW] Yichen Xu, Oliver Bračevac, Cao Nguyen Pham, and Martin Odersky, Artifact for "What's in the Box: Ergonomic and Expressive Capture Tracking over Generic Data Structures" Aug. 2025. doi:10.5281/zenodo.16922930 (cit. on pp. 4, 26).

[55] Yichen Xu, Oliver Bračevac, Cao Nguyen Pham, and Martin Odersky. 2025. What's in the Box: Ergonomic and Expressive Capture Tracking over Generic Data Structures (Extended Version). doi:10.48550/arXiv.2509.07609 (cit. on pp. 10, 12, 13, 16–18).

[56] Yichen Xu and Martin Odersky. 2024. A formal foundation of reach capabilities (extended abstract). In *Companion Proceedings of the International Conference on the Art, Science, and Engineering of Programming (VIMPL'24)*. ACM. doi:10.1145/3660829.3660851 (cit. on p. 3).

[57] Yichen Xu and Martin Odersky. 2023. Formalizing box inference for capture calculus. *ArXiv*, abs/2306.06496 (cit. on p. 6).

Received 2025-03-25; accepted 2025-08-12