Classifying Capabilities (Extended Version)

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

Capture checking in Scala 3 enables lightweight and practical effect and resource tracking by recording capabilities in types. However, the system offers no way to reason about *kinds* of capabilities. Natural constraints such as "retaining only the control-flow capabilities of this closure" or "excluding all thread-local capabilities from this argument" become inexpressible. Both arise in the Scala 3 standard library: `Try` re-throws caught exceptions, so it retains only the control-flow capabilities of its body, and `Future` must not capture thread-local resources. The inability to state these constraints has kept parts of the library outside capture checking.

We introduce *capability classifiers*: a tree-structured, user-extensible hierarchy of tags that classify capabilities by their semantic role. *Projections* filter capture sets by classifier, supporting both inclusion (`c.only[C]`) and exclusion (`c.except[C]`). The tree structure enables decidable disjointness reasoning: classifiers on separate branches are guaranteed to be disjoint regardless of unknown extensions elsewhere in the hierarchy. We formalize classifiers as an extension of System Capless, a core calculus for capture checking, introducing a classifier kind algebra based on intersection, union, and subtraction of classifier subtrees. We extend the operational semantics to model exception interception and establish type safety, effect safety, and handler coverage via a big-step proof, fully mechanized in Lean 4. Classifiers are implemented in the Scala 3 capture checker, and we demonstrate their use on standard library types and real-world effect exclusion patterns.

1 Introduction

Capture checking [6] provides a lightweight discipline for tracking effects as capabilities. A *capability* is a value whose possession grants the authority to perform an effect or access a resource, for example a file handle, a network socket, a mutable reference, or a token that permits throwing an exception. By recording which capabilities a term may use in its *capture set*, the type system enforces lexically scoped bounds on side effects: a function's type signature declares which capabilities it may use, and the compiler statically checks that these bounds are respected. This approach has been integrated into Scala 3, and has been applied to large parts of the Scala standard library, including the collections [59].

Two notable parts of the standard library have raised difficulties when adopting capture checking: exception handling via `Try[T]` and asynchronous computation via `Future[T]`. The difficulty lies not in tracking *individual* capabilities, but in reasoning about *kinds* of capabilities.

Consider `Try[T]` [42]: its constructor executes a closure and catches any thrown exceptions, packaging the result as a success-or-failure value.

```
val result: Try[String] = Try(file.read())
val content: String = result.get // may re-throw IOException
```

The constructor takes its argument by name, so `file.read()` is not evaluated at the call site. Instead, it is implicitly wrapped in a closure that the constructor runs internally. In this example, `Try` catches exceptions thrown by `file.read()` and wraps them as a value. The resulting `Try` is pure with respect to most capabilities, since e.g., the file handle is not retained. However, it is *not* pure with respect to control-flow capabilities such as `CanThrow` [33] labels: calling `.get` may re-throw the caught exception, which exercises that capability. Without a way to express "the control-flow subset of the closure's captures," the type of `Try`'s constructor must be either too permissive (unsoundly claiming purity) or too conservative (retaining all captures).

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

A dual problem arises with `Future[T]`. Its constructor dispatches a closure to a thread pool, where it may execute on an arbitrary carrier thread.

```
boundary: label =>
  Future:            // should reject: label is thread-local
    break(0, label)  // non-local return to another thread's stack
    file.read()
```

As with `Try`, the `Future` constructor takes its body by name, and Scala's colon syntax passes the indented block as that argument. Here, `boundary` introduces a `Label` capability that is tied to the originating thread's stack. Using it inside the `Future` closure would attempt a non-local return across threads, which must be rejected. More generally, capabilities such as thread-local storage, stack-based control flow, and native interop must not leak into the closure. Yet existing capture checking provides no mechanism to *exclude* an entire category of capabilities from a capture set. One can only enumerate specific capabilities as an upper bound.

Both problems share a common root: capture sets track capabilities by identity, but lack the vocabulary to classify capabilities by *kind*. There is no way to filter a capture set to retain only capabilities of a certain kind, nor to exclude all capabilities belonging to another. These are precisely the operations needed to give accurate constraints to `Try` and `Future`, and, as we demonstrate, to many patterns beyond the standard library. The missing abstraction is not better tracking of individual capabilities, but operators that project and exclude whole categories of capabilities.

We propose *capability classifiers*: a tree-structured, user-extensible system of tags that classify capabilities by their semantic role. Each capability is associated with a classifier drawn from a tree-structured universe. Capture sets are enhanced with *projections* that filter by classifier *kinds*, which consist of classifier subtree inclusions (`c.only[C]`) and exclusions (`c.except[C]`). With classifiers, the previously problematic signatures become precise:

```
object Try:
  def apply[T](body: () => T): Try[T]^{body.only[Control]}
object Future:
  def apply[T](body: () -> {any.except[ThreadLocal]} T): Future[T]^{body}
```

That is, `Try` retains only control capabilities from its argument, while `Future` requires its argument to exclude thread-local ones.

The tree structure of classifiers is chosen to fit the setting that Scala-like languages inhabit: an open ecosystem of separately compiled modules. The tree is open, since any classifier can be extended with sub-classifiers in a separate module, yet disjointness remains decidable and stable under such extensions: classifiers on different branches stay disjoint no matter which sub-classifiers other modules add later. Hierarchical classification, filtering, and exclusion each have precedents in effect systems, for instance as boolean algebras [25, 10] or lattices of qualifiers [19]. Our contribution is their integration with capture checking, where the filtered sets contain references to program values rather than abstract effect labels, under the modularity constraints that separate compilation imposes on a language like Scala. We explain this design point in Section 6.1.

Beyond the standard library, we show that classifiers enable many practical patterns: restricting parallel computations to read-only capabilities, modeling privilege qualifiers on transaction objects, and encoding the effect exclusion patterns catalogued by Lutze et al. [25] in real-world codebases. The result is a lightweight extension to capture checking that makes previously awkward library interfaces precise without abandoning the existing capability-based account.

To summarize, our contributions are as follows.

- We introduce capability classifiers for Scala 3 capture checking, motivated by standard-library examples and real-world effect exclusion patterns that require reasoning about capabilities by kind rather than by identity (Section 2).
- We design a tree-structured, user-extensible classifier hierarchy with a decidable kind algebra supporting union, intersection, subtraction, and disjointness, chosen to preserve modular extension under separate compilation (Section 3.1).
- We formalize capture checking with capability classifiers as System Capless($\mathcal{K}$), an extension of System Capless [59] with kind-annotated projections and capture kinding, and prove type safety with a checked big-step semantics, fully mechanized in Lean 4. In contrast to prior capture-checking metatheories based on syntactic progress and preservation, the big-step account lets us reason directly about closure properties of evaluation, yielding not only type and effect safety, but also used label prediction, capture prediction and handler coverage (Sections 3 and 4).
- We implement classifiers in the Scala 3 capture checker, demonstrate them on standard-library types and on real-world effect exclusion patterns, discuss the resulting design decisions, and report on the impact of classifiers in the capture-checked standard library (Sections 5 and 6).

Finally, we review related work in Section 7 and conclude in Section 8.

2 The Case for Classifiers: Capability Kinds in Scala

Capture checking [6, 59, 43] extends Scala's type system to track *capabilities*, informally values "of interest", such as file handles, access-permission tokens, or mutable data structures. In capture checking, types also record which capabilities are used or held, or *captured*, by values of that type. These *capturing types* have the form `T^{x1, ..., xn}`, which includes the shape type `T` and the *capture set* `{x1, ..., xn}`. Capture sets give an upper bound on capabilities that can be accessed by values of the type. A pure type, written `T^{}` or just `T`, cannot capture any capabilities.

By making capability dependencies visible in types, capture checking serves as a lightweight effect system: the capabilities a function closes over precisely describe the effects it may perform, enabling fine-grained control over what each piece of code is allowed to do.

The conventional function type `A => B` is shorthand for `(A -> B)^{any}`[1], a function that may capture arbitrary capabilities. In contrast, the pure function type `A -> B` carries an empty capture set and therefore cannot close over any capability. For example, in the following code, `run` declares that its callback `task` may only use the `net` capability:

```
def main(fs: Filesystem^, net: Network^): Unit =
  def run(task : () ->{net} Unit) = task()
  run(() => net.send(fs.read("secret")))  // error: fs not in {net}
```

Here `task` has type `() ->{net} Unit`, a shorthand for `(() -> Unit)^{net}`. The capture set `{net}` makes explicit that this function may only use the `Network` capability held by `net`. Any attempt to capture `fs`, which has type `Filesystem^` (i.e., `Filesystem^{any}`), is rejected because `fs` is not in the capture set.

Capabilities are extremely versatile: they can express concepts from many different domains, such as exceptions, I/O, mutation, and more. However, capture sets may only mention capability *identities*. This means it is not possible to:

(1) limit entire *kinds* of capabilities based on system specifications or user definitions (e.g., all control-flow-related capabilities);
(2) refer to a *subset* of capabilities captured by a given reference, filtered by such kinds (e.g., only the read-only subset of a mutable data structure); or
(3) exclude capabilities from a capture set.

[1]The universal capability **any** was referred to as **cap** in older works.

```scala
trait ThreadLocal extends Classifier, SharedCapability
trait Control     extends Classifier, ThreadLocal

class CanThrow[-T] extends Control

case class Environment(file: File, exc: CanThrow[Int])

def runOnNewThread(task: () ->{any.except[Control]} Unit)
```

```scala
def f(env: Environment^) =
  // ok: env.file is not Control
  runOnNewThread(() => env.file.read())

  // error: exc is Control
  runOnNewThread:
    given CanThrow[Int] = env.exc
    throw 1
```

Fig. 1. Classifiers in Scala: declarations (left) and excluding the `Control` category with `.except` (right).

At the heart of this problem lies the lack of a classification mechanism for capabilities and the inability to include or exclude them based on that classification. We introduce *classifiers*, a hierarchical system and algebra for classifying capabilities. In this section, we give an informal introduction to classifiers in Scala and showcase the need for them in the standard library.

2.1 Capability Classifiers

A *classifier* is a tag drawn from a tree-structured hierarchy that describes the semantic role of a capability. Each classifier has exactly one parent but may have arbitrarily many children. Because every path from a classifier to the root is unique, a capability can be assigned to exactly one classifier while still being recognized as belonging to every ancestor of that classifier. The root of the tree is the built-in classifier `Capability`, provided by the standard library. The standard library also provides two children of `Capability`: `SharedCapability`, for capabilities safe to share across scopes, and `ExclusiveCapability`, for exclusively owned resources.

Defining classifiers. In Scala, classifiers are represented as traits. A new classifier is introduced by defining a trait that extends the marker trait `Classifier` and a parent classifier (Figure 1, top left). This declares `Control` as a child of `ThreadLocal`. Because classifiers form an open tree, new children can be added to any classifier in a separate module without modifying existing definitions.

Classifying capabilities. A Scala class is associated with a classifier by extending the classifier trait *without* the `Classifier` marker (Figure 1, left). `CanThrow` is a capability [44] that tracks possibly-thrown exceptions of type `T`. Every instance of it is thereby tagged with the `Control` classifier, which lets the type system distinguish it from unrelated capabilities such as file handles. The `-T` syntax declares that `CanThrow` is contravariant: the more precise `T` becomes, the smaller the set of values that can be thrown.

Projections: `.only` and `.except`. Given a capture reference `a`, `a.only[C]` denotes the subset of capabilities reachable through `a` whose classifier is `C` or any descendant of `C`. Dually, `a.except[C]` denotes the subset whose classifier is *not* `C` or any descendant of `C`. These two *projections* also apply to the root capability `any` to constrain an arbitrary capture set by classifier. Projections can be chained: `any.only[SharedCapability].except[Control]` restricts to shared capabilities outside `Control`.

With these tools in hand, consider the example of Figure 1 (right). An environment bundles a file handle and an exception-throwing capability. A function that dispatches work to a new thread must forbid its closure from using thread-local capabilities, such as exception throwing, and the `.except` projection expresses this constraint directly. The capture set `{any.except[Control]}` is polymorphic: it accepts any capability *except* those under `Control`. Accessing the `File` is allowed because files are not `Control`, while throwing exceptions using the `CanThrow` capability [33] is rejected at compile time.

```scala
object Try: /* before classifiers */
  def apply[T](body: () => T): Try[T]

class Try[+T]:
  def get(): T = /* re-throw if needed */
```

```scala
object Try:
  /* with classifiers */
  def apply[T](body: () => T): Try[T]^{body.only[Control]}
```

Fig. 2. `Try[T]` signatures before classifiers (left) and with classifiers (right).

2.2 Case Study: `Try[T]` Captures Control Capabilities

`Try[T]` [42] is a sum type representing either a successful value or an arbitrary runtime exception, enabling fluid conversion between exception-based and monadic error handling. Under previous capture checking, the constructor and re-throwing accessor had the signatures in Figure 2 (left). The constructor `Try.apply` takes a computation with arbitrary captures, yet the resulting `Try[T]` is declared *pure*. At first glance this seems correct: `Try.apply` evaluates `body`, catches any exception, and stores the outcome; it does not retain a reference to the closure itself.

However, this signature overlooks a particular class of capabilities that govern exception throwing. `CanThrow` capabilities [33, 44] track possibly-thrown exceptions through types, and `boundary/break`'s `Label` capability [45] tracks the ability to perform non-local returns. Both are classified under `Control`. When `Try.get` re-throws a stored exception, it is effectively *using* the `Control` capabilities that were in scope when the exception was originally thrown. Therefore `Try[T]` is not truly pure: it retains the `Control` capabilities of the closure. With classifiers, this dependency can be stated precisely (Figure 2, right). The projection `body.only[Control]` selects only the subset of `body`'s capture set whose classifier is `Control` or a descendant of `Control`. Non-control capabilities captured by the closure (e.g., file handles) are excluded from the result type, reflecting the fact that `Try` does not propagate them.

2.3 Case Study: `Future[T]` Cannot Handle Thread-Locals

`Future[T]` [46] represents a computation that may execute on an arbitrary worker thread. Because the task scheduler is free to migrate the computation between carrier threads, the closure submitted to a `Future` must not rely on any thread-local state. Thread-local resources include thread-local storage, local stack manipulation (`Control` capabilities, coroutines), and native C interop.

With classifiers, this restriction is expressed directly in the `Future` constructor:

```scala
object Future:
  def apply[T](body: () ->{any.except[ThreadLocal]} T): Future[T]^{body}
```

The `.except[ThreadLocal]` projection removes any capability whose classifier falls under `ThreadLocal`. Crucially, because exception handling is tied to the thread-local call stack, `Control` is defined as a child of `ThreadLocal`, so `.except[ThreadLocal]` excludes `Control` capabilities as well. This illustrates how the tree structure pays off: a single exclusion of an ancestor classifier automatically covers all of its descendants, without requiring the programmer to enumerate them.

Beyond these standard-library examples, Lutze et al. [25] have demonstrated the practical value of effect exclusions in existing code. We revisit such patterns in Section 6.2.

3 System Capless($\mathcal{K}$): Capture Checking with Classifiers

We formalize capability classifiers in System Capless($\mathcal{K}$). Our work is built upon System Capless [59], a core calculus for capture checking. It models a lambda calculus that tracks capabilities in types. The calculus follows monadic normal form (MNF) [15], where intermediate computations are bound by let expressions and both functions and their arguments are restricted to variables. This does not reduce expressiveness: any application $t\ u$ can be rewritten as $\text{let } x_1 = t \text{ in let } x_2 = u \text{ in } x_1\ x_2$. Types are stratified into *shape types* S (function types, type variables, $\top$) and *capturing types* T,

x, y, z	**Variable**	$s, t, u \coloneqq$	**Term**	$B \coloneqq$	**Capability Bound**
c	**Capture Variable**	x	variable	ϕ	kind
$k, l \in \mathcal{K}$	**Classifier Trait**	v	value	C	capture set
$\phi \coloneqq$	**Classifier Kind**	$x\ y$	application	$E, F \coloneqq$	**Existential Type**
$\emptyset$	empty	$x[S]$	type application	$\exists c : B\,.\,T$	existential
$k - \overline{k_i}$	subtree	$x[c]$	capture application	T	type
$\phi \vee \phi$	union	$\text{let } x = t \text{ in } u$	let	$T, U \coloneqq$	**Type**
$\pi \coloneqq$	**Projected Capture**	$\text{let } \langle c, x \rangle = t \text{ in } u$	existential let	$S\,\hat{}\,C$	capturing
$(\theta \mid \phi)$	projection	$v \coloneqq$	**Value**	S	pure
$C, D \coloneqq \{\pi_1, \cdots, \pi_n\}$	**Capture Set**	$\lambda^{C}(x : T)t$	function	$R, S \coloneqq$	**Shape Type**
$\Gamma \coloneqq$	**Context**	$\lambda^{C}[X <: S]t$	type function	$\top$	top
$\emptyset$	empty	$\lambda^{C}[c : B]t$	capture function	X	type variable
$\Gamma, x : T$	term binding	$\text{pack } \langle C, x \rangle$	pack	$\forall(x : T)E$	function
$\Gamma, X <: S$	type binding	$\theta \coloneqq$	**Capture**	$\forall[X <: S]E$	type function
$\Gamma, c : B$	capture binding	x	variable	$\forall[c : B]E$	capture function
		c	capture variable		

Fig. 3. Abstract syntax of System Capless($\mathcal{K}$). Changes compared to System Capless [59] are marked.

which pair a shape type with a *capture set*. Existential types $\exists c.T$ allow existentially quantifying a capture set, enabling abstraction over unknown capabilities. The calculus supports three forms of polymorphism: term abstraction over values, type abstraction over type variables bounded by shape types, and capture abstraction over capture variables bounded by capture sets. For example, we can type the original `Try[T]` constructor using System Capless as follows:

$$\text{Try.apply} : \forall[T <: \top]\ \forall[c : *]\ \forall(\text{body} : (\forall(u : \text{unit})\ T)\,\hat{}\,\{c\})\ \text{Try}[T]\,\hat{}\,\{\}$$

The constructor is polymorphic in the body's return type T, as well as in its capture set c. The $: *$ bound indicates that the capture set is unbounded: the body can capture any variables in scope. The return type $\text{Try}[T]\,\hat{}\,\{\}$ indicates that the resulting `Try[T]` has no captures, which is incorrect as previously mentioned. We will see how this is correctly typed in System Capless($\mathcal{K}$) in the next section.

The syntax and type system are presented in Figure 3 and Figure 4. Changes compared to System Capless are marked accordingly. System Capless($\mathcal{K}$) is parameterized by a universe of classifiers $\mathcal{K}$, organized into a tree hierarchy. Each base capability and scoped capability can be tagged with one classifier. Captures are enhanced with *projections*, which are filters on classifiers based on a structure called a *classifier kind*.

3.1 Classifiers and Kinds

We embed the classifier tree $\mathcal{K}$ into an infinite tree with root $\top$. $k \sqsubseteq k'$ denotes that k' is an ancestor of k by standard tree relations. Naturally, $k \sqsubseteq \top$ for all classifiers k. Furthermore, for any distinct classifiers k_1, k_2, we know that at most one of $k_1 \sqsubseteq k_2$ and $k_2 \sqsubseteq k_1$ holds. When neither is true, we say that k_1 and k_2 are *disjoint* (i.e., the subtrees rooted at k_1 and k_2 do not share any nodes).

Capture Kinding $\boxed{\Gamma \vdash C : \phi}$

$$\frac{x : S\,\hat{}\,C \in \Gamma \qquad \Gamma \vdash C.\mathrm{proj}[\phi_1] : \phi_2}{\Gamma \vdash \{(x \mid \phi_1)\} : \phi_2}\;(\text{K-VAR}) \qquad \frac{c : C \in \Gamma \qquad \Gamma \vdash C.\mathrm{proj}[\phi_1] : \phi_2}{\Gamma \vdash \{(c \mid \phi_1)\} : \phi_2}\;(\text{K-CVAR}) \qquad \frac{c : \phi_1 \in \Gamma}{\Gamma \vdash \{(c \mid \phi_2)\} : \phi_1 \wedge \phi_2}\;(\text{K-CBOUND})$$

$$\frac{\Gamma \vdash C : \phi \qquad \phi \preceq \phi'}{\Gamma \vdash C : \phi'}\;(\text{K-SUB}) \qquad \frac{\Gamma \vdash C_1 : \phi \qquad \Gamma \vdash C_2 : \phi}{\Gamma \vdash C_1 \cup C_2 : \phi}\;(\text{K-UNION}) \qquad \frac{\phi \textbf{ empty}}{\Gamma \vdash \{(\theta \mid \phi)\} : \phi'}\;(\text{K-ABSURD}) \qquad \Gamma \vdash \{\} : \phi\;(\text{K-EMPTY})$$

Capture Subsetting $\boxed{C_1 \sqsubseteq C_2}$

$$\frac{C_1 \subseteq C_2}{C_1 \sqsubseteq C_2}\;(\text{S-ELEM}) \qquad \frac{\phi \textbf{ empty}}{\{(\theta \mid \phi)\} \sqsubseteq \{\}}\;(\text{S-ABSURD})$$

$$\frac{\phi_1 \preceq \phi_2}{\{(\theta \mid \phi_1)\} \sqsubseteq \{(\theta \mid \phi_2)\}}\;(\text{S-SUBKIND}) \qquad \{(\theta \mid (\phi_1 \vee \phi_2))\} \sqsubseteq \{(\theta \mid \phi_1), (\theta \mid \phi_2)\}\;(\text{S-MERGE})$$

Subcapturing $\boxed{\Gamma \vdash C_1 <: C_2}$ (changed rules; full set in Figure A.1)

$$\frac{x : S\,\hat{}\,C \in \Gamma}{\Gamma \vdash \left\{(x \mid \phi)\right\} <: C.\mathrm{proj}[\phi]}\;(\text{SC-VAR}) \qquad \frac{C_1 \sqsubseteq C_2}{\Gamma \vdash C_1 <: C_2}\;(\text{SC-ELEM})$$

$$\frac{c : C \in \Gamma}{\Gamma \vdash \left\{(c \mid \phi)\right\} <: C.\mathrm{proj}[\phi]}\;(\text{SC-BOUND}) \qquad \frac{\Gamma \vdash C : \phi}{\Gamma \vdash C <: C.\mathrm{proj}[\phi]}\;(\text{SC-PROJ})$$

Bound Subtyping $\boxed{\Gamma \vdash B_1 <:_{\mathrm{B}} B_2}$

$$\frac{\Gamma \vdash C_1 <: C_2}{\Gamma \vdash C_1 <:_{\mathrm{B}} C_2}\;(\text{B-SET}) \qquad \frac{\phi_1 \preceq \phi_2}{\Gamma \vdash \phi_1 <:_{\mathrm{B}} \phi_2}\;(\text{B-KIND}) \qquad \frac{\Gamma \vdash C : \phi}{\Gamma \vdash C <:_{\mathrm{B}} \phi}\;(\text{B-SET-KIND})$$

Typing $\boxed{C; \Gamma \vdash t : E}$ (selected rules; full set in Figure A.1)

$$\frac{x : S\,\hat{}\,C \in \Gamma}{\{x\}; \Gamma \vdash x : S\,\hat{}\,\{x\}}\;(\text{VAR}) \qquad \frac{\begin{array}{c} C'; \Gamma \vdash x : (\forall (z : T) E)\,\hat{}\,C \\ C'; \Gamma \vdash y : T \end{array}}{C'; \Gamma \vdash x\ y : [y/z]E}\;(\text{APP}) \qquad \frac{\begin{array}{c} C; \Gamma \vdash t : T \qquad C; (\Gamma, x : T) \vdash u : E \\ \Gamma \vdash C, E \textbf{ wf} \end{array}}{C; \Gamma \vdash \text{let } x = t \text{ in } u : E}\;(\text{LET})$$

$$\frac{\begin{array}{c} C \uplus \{x\}; (\Gamma, x : T) \vdash t : E \\ \Gamma \vdash T \textbf{ wf} \end{array}}{\{\}; \Gamma \vdash \lambda^C (x : T) t : (\forall (x : T) E)\,\hat{}\,C}\;(\text{ABS}) \qquad \frac{\begin{array}{c} C; \Gamma \vdash x : [D/c]T \\ \Gamma \vdash D <:_{\mathrm{B}} B \end{array}}{\{\}; \Gamma \vdash \text{pack } \langle D, x \rangle : \exists c : B\,.\,T}\;(\text{PACK}) \qquad \frac{\begin{array}{c} C; \Gamma \vdash t : \exists c : B\,.\,T \qquad \Gamma \vdash C, F \textbf{ wf} \\ C; (\Gamma, c : B, x : T) \vdash u : F \end{array}}{C; \Gamma \vdash \text{let } \langle c, x \rangle = t \text{ in } u : E}\;(\text{LET-E})$$

Subtyping $\boxed{\Gamma \vdash E_1 <: E_2}$ (selected rules; full set in Figure A.1)

$$\frac{\begin{array}{c} \Gamma \vdash S_1 <: S_2 \\ \Gamma \vdash C_1 <: C_2 \end{array}}{\Gamma \vdash S_1\,\hat{}\,C_1 <: S_2\,\hat{}\,C_2}\;(\text{CAPT}) \qquad \frac{(\Gamma, c : B_1) \vdash T_1 <: T_2 \qquad \Gamma \vdash B_1 <:_{\mathrm{B}} B_2}{\Gamma \vdash \exists c : B_1 .T_1 <: \exists c : B_2 .T_2}\;(\text{EXIST})$$

Fig. 4. Type system of System Capless($\mathcal{K}$) (condensed). Changes compared to System Capless [59] are marked . The complete type system is given in Figure A.1.

Intersection $\boxed{\phi_1 \wedge \phi_2 = \phi}$ (selected rules; full set in Figure A.2)

$$\frac{k_1 \sqsubseteq k_2}{(k_1 - \overline{l_1}) \wedge (k_2 - \overline{l_2}) = k_1 - (\overline{l_1}, \overline{l_2})} \text{(I-SUBTREE-L)} \qquad \frac{k_1 \perp k_2}{(k_1 - \overline{l_1}) \wedge (k_2 - \overline{l_2}) = \emptyset} \text{(I-DISJOINT)} \qquad \frac{\phi \wedge \phi_1 = R_1 \qquad \phi \wedge \phi_2 = R_2}{\phi \wedge (\phi_1 \vee \phi_2) = R_1 \vee R_2} \text{(I-UNION-R)}$$

Subtraction $\boxed{\phi_1 \setminus \phi_2 = \phi}$ (selected rules; full set in Figure A.2)

$$(k_1 - \overline{l_1}) \setminus (k_2 - \epsilon) = k_1 - (k_2, \overline{l_1}) \text{ (ST-TREE)} \qquad \frac{\phi_1 \setminus (k - \overline{k_i}) = R_1 \qquad \phi_2 \setminus (k - \overline{k_i}) = R_2}{(\phi_1 \vee \phi_2) \setminus (k - \overline{k_i}) = R_1 \vee R_2} \text{(ST-UNION-L)}$$

$$\frac{l \sqsubseteq k_2 \qquad l \sqsubseteq k_1 \qquad (k_1 - \overline{l_1}) \setminus (k_2 - \overline{l_2}) = \phi}{(k_1 - \overline{l_1}) \setminus (k_2 - (l, \overline{l_2})) = (l - \overline{l_1}) \vee \phi} \text{(ST-SUB-R)} \qquad \frac{\phi \setminus \phi_1 = R_1 \qquad R_1 \setminus \phi_2 = R_2}{\phi \setminus (\phi_1 \vee \phi_2) = R_2} \text{(ST-UNION-R)}$$

Emptiness $\boxed{\phi \textbf{ empty}}$

$$\emptyset \textbf{ empty} \text{ (E-EMPTY)} \qquad \frac{k \sqsubseteq l_i}{k - (l_1, \ldots, l_i, \ldots, l_n) \textbf{ empty}} \text{(E-ABSURD)} \qquad \frac{\phi_1 \textbf{ empty} \qquad \phi_2 \textbf{ empty}}{(\phi_1 \vee \phi_2) \textbf{ empty}} \text{(E-UNION)}$$

Disjoint $\boxed{\phi_1 \perp \phi_2}$ Defined as $(\phi_1 \wedge \phi_2)$ **empty**. **Subkinding** $\boxed{\phi_1 \preceq \phi_2}$ Defined as $(\phi_1 \setminus \phi_2)$ **empty**.

Inclusion $\boxed{k \in \phi}$

$$\frac{k \sqsubseteq k_0}{k \in (k_0 - \epsilon)} \text{(IN-BASE)} \qquad \frac{k \not\sqsubseteq k_1 \qquad k \in (k_0 - (k_2, \ldots, k_n))}{k \in (k_0 - (k_1, \ldots, k_n))} \text{(IN-NONEXCL)} \qquad \frac{k \in \phi_i}{k \in (\phi_1 \vee \cdots \vee \phi_i \vee \cdots \vee \phi_n)} \text{(IN-UNION)}$$

Capture Set Projection $C.\mathsf{proj}[\phi] := \left\{\overline{(\theta_i \mid \phi_i \wedge \phi)}\right\}$ where $C = \left\{\overline{(\theta_i \mid \phi_i)}\right\}$

Exclusive Union $C \uplus \{x_1, \ldots, x_n\} = C \cup \{(x_1 \mid \top), \ldots, (x_n \mid \top)\}$ where $\forall i, \phi.(x_i \mid \phi) \notin C$

Fig. 5. Kind algebra of System Capless($\mathcal{K}$) (condensed). The complete kind algebra is given in Figure A.2.

Each node has a countably infinite number of children. This is crucial for modularity, as a classifier cannot be “singled out” by excluding all its known sub-classifiers: additional sub-classifiers can be added in another module, as in the following example:

```
// Module 1
trait A extends Classifier, SharedCapability
trait B extends Classifier, A // C likewise
def f(body: () ->{any.only[A].except[B].except[C]} Unit)
```

```
// Module 2, compiled separately
trait D extends Classifier, A
```

Function `f` cannot assume that no sub-classifier of `A` exists, since it can be arbitrarily extended by another module. Therefore, classifiers naturally avoid the closed-world problem presented by Lutze et al. [25]. Flix overcomes this problem by treating the effect set top as a symbolic constant during boolean unification, preventing negations from being translated into a finite positive set. In System Capless($\mathcal{K}$), each subtree is assumed to have infinitely many children, so this cannot happen.

Kinds ϕ represent filters for classifiers. To model `.only` and `.except`, we work with subtrees of classifiers when constructing kinds. They are represented as unions of “holed classifier subtrees”: a root subtree k_0 and a list of excluded subtrees $k_1, \ldots, k_n$. Each such subtree represents the set of all classifiers k that are sub-classifiers of k_0 but are *not* sub-classifiers of any remaining k_i (i.e., $k \sqsubseteq k_0$

Semantic Domains

$s, t, u ::= \dots$	**Term**	$\theta ::= \dots$	**Capture**	$\Sigma ::=$	**Label Context**	$w ::=$	**Return Value**
ℓ	label	ℓ	label	$\emptyset$	empty	v	return
$v ::= \dots$	**Value**	$a ::= \langle \Sigma, w \rangle$	**Answer**	$\Sigma,\ \ell : S :: k$	binding	$\mathsf{brk}(\ell, v)$	break
ℓ	label					$F ::= \{\ell_1, \dots, \ell_n\}$	**Label Set**

Potential Use Set $\boxed{\lfloor v \rfloor}$

$$\lfloor \lambda^D(x : T)t \rfloor = D \qquad \lfloor \mathsf{pack}\ \langle C, v \rangle \rfloor = \{\}$$
$$\lfloor \lambda^D[X <: S]t \rfloor = D \qquad \lfloor \ell \rfloor = \{\ell\}$$
$$\lfloor \lambda^D[c : B]t \rfloor = D$$

Runtime Labels $\boxed{C^*_\Sigma}$ $\boxed{v^*_\Sigma}$

$$\emptyset^*_\Sigma = \emptyset \qquad (C_1 \cup C_2)^*_\Sigma = (C_1)^*_\Sigma \cup (C_2)^*_\Sigma$$
$$\{(\theta \mid \phi)\}^*_\Sigma = \begin{cases} \{\ell\} & \theta = \ell,\ (\ell : S :: k) \in \Sigma \text{ and } k \in \phi \\ \emptyset & \text{otherwise} \end{cases}$$
$$v^*_\Sigma = \lfloor v \rfloor^*_\Sigma$$

Big-step Evaluation (Selected) $\boxed{\Sigma \vdash t \Downarrow_F a}$

$$\frac{}{\Sigma \vdash v \Downarrow_F \langle \Sigma, v \rangle} \quad \text{(E-VAL)}$$

$$\frac{\Sigma \vdash t_1 \Downarrow_F \langle \Sigma', v \rangle \qquad \Sigma' \vdash [\lfloor v \rfloor / x][v/x]t_2 \Downarrow_F a}{\Sigma \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \Downarrow_F a} \quad \text{(E-LET-V)}$$

$$\frac{\Sigma \vdash t_1 \Downarrow_F \langle \Sigma', \lambda^D(x : T)t \rangle \qquad \Sigma' \vdash t_2 \Downarrow_F \langle \Sigma'', v \rangle \qquad D^*_{\Sigma''} \subseteq F \qquad v^*_{\Sigma''} \subseteq F \qquad \Sigma'' \vdash [\lfloor v \rfloor / x][v/x]t \Downarrow_{D^*_{\Sigma''} \cup v^*_{\Sigma''}} a}{\Sigma \vdash t_1 t_2 \Downarrow_F a} \quad \text{(E-APP)}$$

$$\frac{\Sigma \vdash t_1 \Downarrow_F \langle \Sigma', \mathsf{pack}\ \langle D, v \rangle \rangle \qquad \Sigma' \vdash [D/c][\lfloor v \rfloor / x][v/x]t_2 \Downarrow_F a}{\Sigma \vdash \mathsf{let}\ \langle c, x \rangle = t_1\ \mathsf{in}\ t_2 \Downarrow_F a} \quad \text{(E-EX-V)}$$

$$\frac{\Sigma \vdash t_1 \Downarrow_F \langle \Sigma', \ell \rangle \qquad \Sigma' \vdash t_2 \Downarrow_F \langle \Sigma'', v \rangle \qquad \ell \in F}{\Sigma \vdash t_1 t_2 \Downarrow_F \langle \Sigma'', \mathsf{brk}(\ell, v) \rangle} \quad \text{(E-BREAK)}$$

$$\frac{\Sigma \vdash t \Downarrow_F \langle \Sigma', \lambda^D[X <: S_0]t \rangle \qquad D^*_{\Sigma'} \subseteq F \qquad \Sigma' \vdash [S/X]t \Downarrow_{D^*_{\Sigma'}} a}{\Sigma \vdash t[S] \Downarrow_F a} \quad \text{(E-TAPP)}$$

$$\frac{\Sigma \vdash t_1 \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell, v) \rangle}{\Sigma \vdash t_1 t_2 \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell, v) \rangle} \quad \text{(E-APP-B1)}$$

$$\frac{\Sigma \vdash t \Downarrow_F \langle \Sigma', \lambda^D[c : B]t \rangle \qquad D^*_{\Sigma'} \subseteq F \qquad \Sigma' \vdash [C/c]t \Downarrow_{D^*_{\Sigma'}} a}{\Sigma \vdash t[C] \Downarrow_F a} \quad \text{(E-CAPP)}$$

$$\frac{\Sigma \vdash t_1 \Downarrow_F \langle \Sigma', v_1 \rangle \qquad \Sigma' \vdash t_2 \Downarrow_F \langle \Sigma'', \mathsf{brk}(\ell, v) \rangle}{\Sigma \vdash t_1 t_2 \Downarrow_F \langle \Sigma'', \mathsf{brk}(\ell, v) \rangle} \quad \text{(E-APP-B2)}$$

Fig. 6. Checked big-step semantics of System Capless($\mathcal{K}$): selected core rules. Remaining rules handle breaks in other constructs and can be found in Figure A.4.

and $\forall i > 0, k \not\sqsubseteq k_i$). The inclusion relation ($k \in \phi$) determines whether a classifier is included in a classifier kind. It is decidable, and is also used in the operational semantics.

We present in Figure 5 algorithmic rules for intersection ($\phi_1 \wedge \phi_2$) and subtraction ($\phi_1 \setminus \phi_2$) of kinds. The unambiguity of the rules depends on the previously mentioned sub-classifier-or-disjoint property of the classifier tree.

The intersection of two subtrees can simply be computed as a new subtree whose exclusion list contains both exclusion lists of the two inputs, and the new root is the deepest common classifier of the two input roots. This is the smaller root if they have a sub-classifier relationship ((I-SUBTREE-L) and (I-SUBTREE-R)), or none if the roots are disjoint (I-DISJOINT). Compound cases are handled by computing the union of all intersections of pairs of subtrees.

Similarly, we handle subtraction of two subtrees individually, and compound cases are handled by the distribution on the left-hand side $(\phi_1 \vee \phi_2) \setminus \phi = (\phi_1 \setminus \phi) \vee (\phi_2 \setminus \phi)$ (ST-UNION-L) and composition on the right-hand side $\phi \setminus (\phi_1 \vee \phi_2) = (\phi \setminus \phi_1) \setminus \phi_2$ (ST-UNION-R). For subtrees, we compute recursively on the exclusion list of the right-hand side. In the empty case (ST-TREE), we can simply append the root to the left-hand side's exclusion list. Otherwise, we follow the general equality of set subtractions $A \setminus (B \setminus C) = (A \setminus B) \cup (A \cap B \cap C)$. The rules (ST-IRREL-R), (ST-ABSURD-R), (ST-IRREL-L), (ST-SUB-L), and (ST-SUB-R) handle each case based on the sub-classifier/disjointness relationship between the excluded node and the two roots.

Subkinding ($\phi_1 \preceq \phi_2$) can be defined as empty subtraction ($(\phi_1 \setminus \phi_2)$ **empty**), and disjointness of kinds can be defined as empty intersection. This is grounded in the fact that the subtraction operator satisfies the axioms of subtraction algebras [48, 16], which define an ordering relation.

There can be multiple representations of the empty set, so ϕ **empty** is also defined algorithmically. The interesting case is when a subtree contains an ancestor node in its exclusion list (E-ABSURD).

We use a single classifier name to represent a subtree with the given classifier as root and no excluded roots, as well as a classifier kind containing that sole subtree. Following this notation, $\top$ is the classifier kind containing all classifiers. The kind representing all classifiers except a subtree k can be written as $\top - k$.

The definitions of intersection and subtraction are equivalent to the set semantics. More details about the proofs can be found in Section 4.1.

3.2 Capability Projections

At the core of System Capless($\mathcal{K}$) lie *projected captures* $(\theta \mid \phi)$. A projected capture consists of a capture θ, applied to a classifier kind filter ϕ. It represents *sub-parts* of θ whose classifier k matches the kind ϕ (i.e., $k \in \phi$). For instance, we can correctly type the signature of the `Try[T]` constructor (previously mentioned in Section 2.2) in System Capless($\mathcal{K}$) as

$$\text{Try.apply} : \forall[T <: \top]\ \forall[c : \top]\ \forall(\text{body} : (\forall(u : \text{unit})\ T)\,\hat{}\,\{c\})\ \text{Try}[T]\,\hat{}\,\{(c \mid \text{Control})\}$$

The return type captures a projection of the `body`'s capture set c, filtered to only the classifier subtree rooted at `Control`. The unbounded $*$ is replaced with the $\top$ kind bound, allowing all capture sets. Note that the capture set $\{(c \mid \text{Control})\}$ is attached to the return type and not the function: the constructor is *pure* and does not capture any capabilities by itself.

For brevity we omit projections to $\top$. The notation $C.\text{proj}[\phi]$ applies a further projection to each element in C by intersecting ϕ with each singleton's existing projection, i.e.,

$$\{(\theta_1 \mid \phi_1), \ldots, (\theta_n \mid \phi_n)\}.\text{proj}[\phi] = \{(\theta_1 \mid \phi_1 \wedge \phi), \ldots, (\theta_n \mid \phi_n \wedge \phi)\}$$

Capture subsetting $C_1 \sqsubseteq C_2$ follows normal subsetting rules $C_1 \subseteq C_2$, similar to System Capless [59], but is also extended to handle projections. A projected capture is smaller than the same capture but with a super-kind (i.e., a coarser filter), by (S-SUBKIND). Projection to an empty kind is equivalent to removing that capture from the set (S-ABSURD). Finally, (S-MERGE) allows us to treat one capture projected to a union of two kinds as equivalent to two separate projected captures.

In System Capless, capture set variables can be either unbounded (as with the `Try[T]` example above) or bounded by other capture sets. System Capless($\mathcal{K}$) replaces unbounded capture set variables with *kind-bounded* variables: they can only be instantiated with capture sets satisfying the given kind. For example, the constructor for `Future[T]` (Section 2.3), which forbids capabilities under `ThreadLocal`, can be typed as:

$$\text{Future.apply} : \forall[T <: \top]\ \forall[c : \top - \text{ThreadLocal}]\ \forall(\text{body} : (\forall(u : \text{unit})\ T)\,\hat{}\,\{c\})\ \text{Future}[T]\,\hat{}\,\{c\}$$

where the kind $\top - \text{ThreadLocal}$ represents the kind of all classifiers, except the subtree rooted at `ThreadLocal`. The type system governs which capture sets can be instantiated under a kind bound through the *capture kinding* process described in the next section.

3.3 Capture Kinding and Subcapturing

Subcapturing rules $C_1 <: C_2$ largely mirror those of System Capless. They follow the subsetting relation $C_1 \sqsubseteq C_2$, but with additional rules for term (SC-VAR) and capture variables (SC-BOUND), which refine captures based on their bounds. Projections on the variables are applied to the corresponding bound capture sets. The only new rule is (SC-PROJ), which allows us to equate any capture set C to its projected version $C.\mathsf{proj}[\phi]$, if we can deduce that C contains only captures under kind ϕ, through a judgment called *capture kinding*, which we shall describe below.

The capture kinding judgment $\Gamma \vdash C : \phi$ is derived by looking up each projected capture in the context. For capture-set-bounded term and capture variables, we recursively check the definitions (in (K-VAR) and (K-CVAR)). Similar to subcapturing, projections of the capture are applied to the upper bound. For kind-bounded capture set variables in (K-CBOUND), the intersection of the projection and the kind bound is the most precise kind we can derive. Subkinding is handled by (K-SUB), with a special case for empty kinds in (K-ABSURD) to allow kinding judgments even on invalid references with empty projections.

Note that the simple addition of (SC-PROJ) allows us to derive useful, non-trivial facts. That projection preserves subcapturing ($\Gamma \vdash C <: D$ implies $\Gamma \vdash C.\mathsf{proj}[\phi] <: D.\mathsf{proj}[\phi]$) can be proven by induction on the original derivation and using composition of projections ($C.\mathsf{proj}[\phi_1].\mathsf{proj}[\phi_2] = C.\mathsf{proj}[\phi_1 \wedge \phi_2]$). Empty-kinded sets can be proven to be empty ($\Gamma \vdash C : \phi$ where ϕ **empty** implies $\Gamma \vdash C <: \{\}$) by first projecting C to the empty kind ϕ using (SC-PROJ), then using (SC-ELEM) to conclude $\Gamma \vdash C.\mathsf{proj}[\phi] <: \{\}$, where the underlying subset relation is obtained by repeatedly applying (S-ABSURD) to each projected capture.

Together, kinding and subcapturing allow refinement and filtering of variables in capture sets. For example, with two disjoint child classifiers $k_1, k_2 \sqsubseteq k$ and a kind-bounded capture set variable $c : k_1$ in scope, we can derive that $\{c\}.\mathsf{proj}[k_2] <: \{\}$, i.e., a function capturing values of kind k_2 will not capture any capabilities of kind k_1.

3.4 Typing and Subtyping

Subtyping remains unchanged from System Capless [59]: subtyping of capturing types $S_1 \wedge C_1 <: S_2 \wedge C_2$ requires both subtyping on the shape types and subcapturing on the capture sets (CAPT), but otherwise follows kernel System $\mathrm{F}_{\leq}$, unsurprisingly. Capture set bound subtyping either follows subcapturing (when both bounds are capture sets), subkinding (when both bounds are kinds), or capture kinding when we refine a kind bound to a capture-set bound (B-SET-KIND).

Similar to System Capless, the typing judgment $C; \Gamma \vdash t : E$ states that t has the existential type E under context Γ with the *use set* C. The use set represents capabilities that *may* be used during the evaluation of t. We follow System Capless for use set judgments: values can be typed with an empty use set as they are already evaluated, variables are typed with themselves in the use set (VAR), and abstractions (ABS), (TABS), (CABS) reify the body's use set into the function's *capture set*, leaving an empty use set. Note that the use set is recorded in the lambda term, which is used in the checked big-step semantics (Section 3.6). Otherwise, typing is standard with regard to System $\mathrm{F}_{\leq}$, and remains unchanged from System Capless. Note that let typing (LET) and (LET-E) require the final use set C and return types to be well-formed in the absence of the let-bound parameter, which we can achieve by widening its appearances in capture sets (e.g., by applying (SC-VAR)).

3.5 Translation from Surface Language

Scala provides `any` as syntactic sugar for the *universal capability*: a reference standing for "some unspecified capability in scope". It does not appear in the formal calculus. Instead, `any` is *desugared* into explicit quantification over a fresh capture variable. In a *contravariant* input position, `any` introduces a fresh *capture abstraction*: the enclosing function becomes capture-polymorphic. In a *covariant* (output) position, `any` becomes an *existential type*. Consider the `Try.apply` constructor from Section 2.2:

```
def apply[T](body: () ->{any} T): Try[T]^{body.only[Control]}
```

The contravariant `any` on `body`'s type introduces a fresh capture variable $c : \top$. It is translated to:

$$\mathrm{Try.apply} : \forall[T <: \top]\ \forall[c : \top]\ \forall(\mathrm{body} : (\forall(u : \mathrm{unit})\ T)\,\hat{}\,\{c\})\ \mathrm{Try}[T]\,\hat{}\,\{(c \mid \mathrm{Control})\}$$

The translation between the surface language and System Capless has been formalized in System Reacap [59], a surface calculus for capture checking with `any`, and applies directly to System Capless($\mathcal{K}$). Section 5 describes the implementation.

3.6 Big-Step Semantics

We define *checked* big-step semantics in Figures 6 and 8. The judgment $\Sigma \vdash t \Downarrow_F a$ includes access checks (in green) under label allowance F. The big-step evaluation involves a label context Σ that records the classifier and value type associated with created labels throughout the evaluation. Therefore, evaluation always returns a possibly-appended label context as part of its answer: either a return value $\langle \Sigma', v \rangle$ or a propagating break $\langle \Sigma', \mathrm{brk}(\ell, v) \rangle$.

Ignoring all access checks, the judgments ((E-VAL), (E-APP), (E-TAPP), (E-CAPP), (E-LET-V), (E-EX-V)) describe a standard top-level (i.e., no store bindings), substitution-based, call-by-value big-step reduction over the *relaxed*, direct-style syntax used in the proofs (Figure A.3), since substitution takes evaluation out of MNF. Note that substituting a term variable x involves both a capture-set substitution $[C/x]$ and a regular term substitution $[v/x]$, as x can appear in both contexts.

The access checks make the scoping discipline of labels explicit. The allowance F directly governs breaks: invoking a label (E-BREAK) is stuck if the label is not included in the allowance. At the outer typing-to-runtime boundary, the allowance is the runtime label set computed from the static typing capture set: it records only which labels may be accessed during evaluation, by comparing each label's projection kind to the current label context. Within the rules, allowances are otherwise ordinary runtime label sets. Intuitively, the runtime labels v^*_Σ (Figure 6) of a λ-abstraction are the set of labels that it can reach when its body is evaluated. Accordingly, λ-abstractions are entered under the label allowance dictated by their annotated capture sets.[2] Application rules ((E-APP), (E-TAPP), (E-CAPP)) check that the abstractions' and their arguments' runtime labels are bounded by the current allowance. This is what allows propagated breaks to stay well-scoped. All other rules support the *extended* System Capless($\mathcal{K}$), which we describe in the next section.

3.7 Boundary, Break, and Intercept

To prove safety properties about the system (Section 4), we extend System Capless($\mathcal{K}$) with additional control-flow constructs (Figure 7): *boundary*, *break* (following [59, 6]), and a new construct *intercept*.

Boundary and Break. Following [59, 6], $\mathrm{boundary}[S, k]$ as $\langle c, x \rangle$ in t introduces a scope with a fresh label ℓ, which the body can use to transfer control (*break*) disruptively out of the scope. We extend this with a classifier k: the boundary rules ((E-BND-V), (E-BND-C), (E-BND-P)) create a fresh label with an arbitrary sub-classifier $k_0 \sqsubseteq k$. This non-determinism models the runtime class of a caught exception, which may be a concrete subclass of the declared one.

Breaks can be invoked by applying a label value l (of type $\mathrm{Break}\ [S]$) to the appropriate value v of type S ((E-BREAK) and statically (T-BREAK)), returning the disruptive answer $\mathrm{brk}(\ell, v)$. This

[2] We assume that λ-abstractions carry their annotated capture sets (cf. Figure 4).

is propagated upwards, in a similar way to throwing exceptions (for example, as described by (E-APP-B1) and (E-APP-B2) for applications; other propagation rules can be found in A.4), until it is either caught by the corresponding boundary (E-BND-C) or an intercept (described below), or reaches the top level. Boundaries push the fresh label onto the persistent label context, but the label allowance includes it only during the evaluation of the body: the label may escape (e.g., inside returned abstractions), but invoking an escaped label is stuck. Corollary 4.3 proves that well-typed closed terms do not evaluate to brk.

Intercept. The construct intercept$[S, C_t, \phi]$ with h in t intercepts any break that escapes t whose label has classifier $k \in \phi$, dispatching the caught label and its payload to the handler h. Evaluation is deterministic: a matching break is handled, a non-matching one propagates, and a normal return passes through unchanged ((E-ICP-M), (E-ICP-P), (E-ICP-V)). The inner term runs under the allowance of its annotated use set and may thus invoke more labels than the context allows, as long as they are caught: handlers are evaluated with the original allowance (E-ICP-M), while propagated labels are re-checked to be in the same allowance (E-ICP-P). Interestingly, we can refine the use set of the intercept expression: after handling labels of kind ϕ, only captures outside ϕ may escape from the body, together with the handler's own captures. Specifically, for handlers that *do not* re-throw (i.e., those that do not use the captured label), we can drop matching labels from the use set of t, so that only $C_t.\mathsf{proj}[\top \setminus \phi]$ is included (T-ICP-PASS).

The intercept construct acts as a modular extension to the extended System Capless [59]. It is a straightforward model for *intercepting* exception handlers: handlers that intercept an existing lexical exception handler, as in the case of Scala's interactions between `Try[T]` [42] and `boundary` [45]. The separation is meaningful: it precisely defines the requirements that intercepting handlers must uphold to maintain *scope-safety*, without modifying the existing assumptions of lexical handlers.

Encoding `Try[T]`. Scala's `Try[T]` can be encoded into the extended System Capless($\mathcal{K}$). Using Church encoding, we can represent the two cases of the container (a value of type `T`, or a brk of unknown type) as follows, where the extra capture-set parameter C represents the capture set of the Try value itself, to appear within parameter positions of the handlers:

$$\begin{aligned}\mathsf{Try}(T, C) \;=\; & \forall[X <: \top].\ \forall[c : \top]. \\ & \Big(\forall (s : (\forall(x : T)\, X)\, \hat{}\, \{c\}). \\ & \quad \forall (f : (\forall[Y <: \top].\ \forall(b : (\mathsf{Break}\ [Y])\, \hat{}\, C)\ (\forall(r : Y)\, X)\, \hat{}\, \{c\})\, \hat{}\, \{c\}).\ X\Big)\, \hat{}\, \{c : \top\} \cup C\end{aligned}$$

The constructor can then be encoded as taking a thunk b, running it under an intercept where $\phi = \mathsf{Control}$, with the handler and final results wrapped accordingly into different cases of the $\mathsf{Try}(T, \{(b \mid \mathsf{Control})\})$. The full development in Lean 4 is further explained in Appendix B.9.

4 Metatheory

We establish soundness of System Capless($\mathcal{K}$) in three steps. First, the classifier kind algebra admits a set-theoretic semantics that justifies extensional reasoning about projections and handler dispatch. Second, unlike prior work on capture calculi [6, 59], which establishes soundness by small-step preservation and progress, we use a checked big-step semantics, a first in that line of work. This is useful for two reasons: (1) it makes the scoping discipline of labels explicit, and (2) it lets us reason directly about what values reach and how this relates to capture sets and classifiers. Third, we employ a standard safety proof for big-step semantics, with relaxed typing rules to handle direct-style, non-MNF terms, and an extension to answer typing that links returned breaks to label scoping, which allows us to show the relation between static typing capture/use sets and the usage of labels at runtime. The closest precedent is the second-class values literature [35, 55, 52], which

Syntax

$s, t, u, h \coloneqq \dots$		**Term**
	$\mathsf{boundary}[S, k]\ \mathsf{as}\ \langle c, x\rangle\ \mathsf{in}\ t$	boundary
	$\mathsf{intercept}[R, C_t, \phi]\ \mathsf{with}\ h\ \mathsf{in}\ t$	intercept
$R, S \coloneqq \dots$		**Shape Type**
	$\mathsf{Break}\,[S]$	break

Subtyping $\boxed{\Gamma \vdash E_1 <: E_2}$

$$\frac{\Gamma \vdash S_2 <: S_1}{\Gamma \vdash \mathsf{Break}\,[S_1] <: \mathsf{Break}\,[S_2]} \quad \text{(BREAK)}$$

Exception Handler Type Abbreviations

$$\mathsf{Handler}_{\mathsf{pass}}[E, \phi, C_h, C_t] \;:=\; \forall[X{<:}\top]\, \forall[c : C_t.\mathsf{proj}[\phi]]\, \forall(b : \mathsf{Break}\,[X]\,\hat{}\,\{c\})\, (\forall(y : X)\, E)\,\hat{}\,C_h$$

$$\mathsf{Handler}_{\mathsf{gen}}[E, \phi, C_h, C_t] \;:=\; \forall[X{<:}\top]\, \forall[c : C_t.\mathsf{proj}[\phi]]\, \forall(b : \mathsf{Break}\,[X]\,\hat{}\,\{c\})\, (\forall(y : X)\, E)\,\hat{}\,C_h \cup \{c\}$$

Typing $\boxed{C; \Gamma \vdash t : E}$

$$\frac{C; \Gamma \vdash x : \mathsf{Break}\,[S]\,\hat{}\,C_x \qquad C; \Gamma \vdash y : S \qquad \Gamma \vdash R\ \textsc{wf}}{C; \Gamma \vdash x\ y : R} \quad \text{(T-BREAK)}$$

$$\frac{C_t; \Gamma \vdash t : E \qquad C_h; \Gamma \vdash h : \mathsf{Handler}_{\mathsf{pass}}[E, \phi, C_h, C_t]}{C_t.\mathsf{proj}[\top \setminus \phi] \cup C_h; \Gamma \vdash \mathsf{intercept}[E, C_t, \phi]\ \mathsf{with}\ h\ \mathsf{in}\ t : E} \quad \text{(T-ICP-PASS)}$$

$$\frac{C \uplus \{c, x\}; (\Gamma, c : k, x : \mathsf{Break}\,[S]\,\hat{}\,\{c\}) \vdash t : S \qquad \Gamma \vdash S\ \textsc{wf}}{C; \Gamma \vdash \mathsf{boundary}[S, k]\ \mathsf{as}\ \langle c, x\rangle\ \mathsf{in}\ t : S} \quad \text{(T-BND)}$$

$$\frac{C_t; \Gamma \vdash t : E \qquad C_h; \Gamma \vdash h : \mathsf{Handler}_{\mathsf{gen}}[E, \phi, C_h, C_t]}{C_t \cup C_h; \Gamma \vdash \mathsf{intercept}[E, C_t, \phi]\ \mathsf{with}\ h\ \mathsf{in}\ t : E} \quad \text{(T-ICP-GEN)}$$

Fig. 7. Extended System Capless($\mathcal{K}$) with boundaries and interceptions.

also reasons in big-step style about what function values may capture, and logical-relations work on reachability types [3], which connects annotations to the values reachable at runtime. Neither qualifies types by classifier-indexed capture sets governed by subcapturing, nor tracks delimiter-scoped control. We sketch the most important aspects here and refer to the Lean mechanization (Appendix B) for the full development.

4.1 Set Semantics of Classifier Kinds

The kind algebra (Figure 5) admits a set-theoretic reading. Define $[\![\phi]\!] = \{k \mid k \in \phi\}$, where membership is the decidable inclusion judgment of Figure 5. Then intersection, subtraction, emptiness, sub-kinding, and disjointness correspond exactly to set intersection, difference, emptiness, inclusion, and disjointness. This lets us carry out later arguments about classifier projection, exclusion, and handler dispatch by ordinary set reasoning on kind denotations.

4.2 Runtime Relaxed Typing

Evaluation introduces labels and direct-style, non-MNF terms, which are not included in the original System Capless($\mathcal{K}$) static typing. To reason about runtime terms, we introduce *relaxed typing* judgments $C; \Sigma; \Gamma \vdash_\star t : E$, which are additionally parameterized by the label context and relax the MNF restrictions. Relaxed typing rules largely follow regular typing rules. We present the most interesting differences. The full set of rules can be found in Figure A.3.

The two judgments are connected by type-preserving translations to and from MNF: any typed MNF term ($C; \Gamma \vdash t : E$) is relax-typed as is ($C; \Sigma; \Gamma \vdash_\star t : E$), and MNF normalization (Definition A.3) maps any relax-typed term back to the original typing with label rules added ($C; \Sigma; \Gamma \vdash \widehat{t} : E$). Therefore, relaxed typing works as a proof device: our safety theorem (Theorem 4.1) maintains

Big-step Evaluation (continued) $\boxed{\Sigma \vdash t \Downarrow_F a}$

$$\frac{\ell \text{ fresh} \qquad k_0 \sqsubseteq k \qquad t' = [\{\ell\}/c][\{\ell\}/x][\ell/x]t \qquad \Sigma, \ell : S :: k_0 \vdash t' \Downarrow_{F\cup\{\ell\}} \langle \Sigma', v\rangle}{\Sigma \vdash \mathsf{boundary}[S, k] \text{ as } \langle c, x\rangle \text{ in } t \Downarrow_F \langle \Sigma', v\rangle} \text{(E-BND-V)}$$

$$\frac{\ell \text{ fresh} \qquad k_0 \sqsubseteq k \qquad t' = [\{\ell\}/c][\{\ell\}/x][\ell/x]t \qquad \Sigma, \ell : S :: k_0 \vdash t' \Downarrow_{F\cup\{\ell\}} \langle \Sigma', \mathsf{brk}(\ell, v)\rangle}{\Sigma \vdash \mathsf{boundary}[S, k] \text{ as } \langle c, x\rangle \text{ in } t \Downarrow_F \langle \Sigma', v\rangle} \text{(E-BND-C)}$$

$$\frac{\ell \text{ fresh} \qquad k_0 \sqsubseteq k \qquad t' = [\{\ell\}/c][\{\ell\}/x][\ell/x]t \qquad \Sigma, \ell : S :: k_0 \vdash t' \Downarrow_{F\cup\{\ell\}} \langle \Sigma', \mathsf{brk}(\ell', v)\rangle \qquad \ell \neq \ell'}{\Sigma \vdash \mathsf{boundary}[S, k] \text{ as } \langle c, x\rangle \text{ in } t \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell', v)\rangle} \text{(E-BND-P)}$$

$$\frac{\Sigma \vdash t \Downarrow_{(C_t)^*_\Sigma} \langle \Sigma', v\rangle}{\Sigma \vdash \mathsf{intercept}[R, C_t, \phi] \text{ with } h \text{ in } t \Downarrow_F \langle \Sigma', v\rangle} \text{(E-ICP-V)}$$

$$\frac{\Sigma \vdash t \Downarrow_{(C_t)^*_\Sigma} \langle \Sigma', \mathsf{brk}(\ell, v)\rangle \qquad (\ell : S :: k) \in \Sigma' \qquad k \in \phi \qquad \Sigma' \vdash h[S][\{(\ell \mid \phi)\}]\ \ell\ v \Downarrow_F a}{\Sigma \vdash \mathsf{intercept}[R, C_t, \phi] \text{ with } h \text{ in } t \Downarrow_F a} \text{(E-ICP-M)}$$

$$\frac{\Sigma \vdash t \Downarrow_{(C_t)^*_\Sigma} \langle \Sigma', \mathsf{brk}(\ell, v)\rangle \qquad (\ell : S :: k) \in \Sigma' \qquad k \notin \phi \qquad \ell \in F}{\Sigma \vdash \mathsf{intercept}[R, C_t, \phi] \text{ with } h \text{ in } t \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell, v)\rangle} \text{(E-ICP-P)}$$

Fig. 8. Checked big-step semantics of System Capless($\mathcal{K}$): boundary and intercept rules.

Capture Kinding (for labels) $\boxed{\Sigma; \Gamma \vdash C : \phi}$

$$\frac{(\ell : S :: k) \in \Sigma \qquad k \in \phi_l \qquad k \in \phi}{\Sigma; \Gamma \vdash \{(\ell \mid \phi_l)\} : \phi} \text{(K-LABEL)} \qquad \frac{(\ell : S :: k) \in \Sigma \qquad k \notin \phi_l}{\Sigma; \Gamma \vdash \{(\ell \mid \phi_l)\} : \emptyset} \text{(K-LABEL-ABSURD)}$$

Relaxed Typing (Selected Rules) $\boxed{C; \Sigma; \Gamma \vdash_\star t : E}$

$$\frac{(\ell : S :: k) \in \Sigma \qquad k \in \phi}{\emptyset; \Sigma; \Gamma \vdash_\star \ell : \mathsf{Break}\ [S]\,\hat{}\,\{(\ell \mid \phi)\}} \text{(RT-LABEL)} \qquad \frac{C'; \Sigma; \Gamma \vdash_\star x : [D/c]T \qquad \Sigma; \Gamma \vdash D <:_{\mathrm{B}} B}{\emptyset; \Sigma; \Gamma \vdash_\star \mathsf{pack}\ \langle D, x\rangle : \exists c : B.T} \text{(RT-PACK-VAR)}$$

$$\frac{C_1; \Sigma; \Gamma \vdash_\star t_1 : (\forall (z : S\,\hat{}\,C_0)E)\,\hat{}\,C_1 \qquad C_2; \Sigma; \Gamma \vdash_\star t_2 : S\,\hat{}\,C_2 \qquad \Sigma; \Gamma \vdash C_2 <: C_0}{C_1 \cup C_2; \Sigma; \Gamma \vdash_\star t_1\ t_2 : [C_2/z]E} \text{(RT-APP)} \qquad \frac{\emptyset; \Sigma; \Gamma \vdash_\star v : [D/c]T \qquad \Sigma; \Gamma \vdash D <:_{\mathrm{B}} B}{\emptyset; \Sigma; \Gamma \vdash_\star \mathsf{pack}\ \langle D, v\rangle : \exists c : B.T} \text{(RT-PACK-VAL)}$$

Answer Typing $\boxed{F; \Sigma; \Gamma \vDash_{\mathrm{a}} w : E}$

$$\frac{\emptyset; \Sigma; \Gamma \vdash_\star v : E}{F; \Sigma; \Gamma \vDash_{\mathrm{a}} v : E} \text{(SA-RET)} \qquad \frac{(\ell : S :: k) \in \Sigma \qquad \ell \in F \qquad \emptyset; \Sigma; \Gamma \vdash_\star v : S}{F; \Sigma; \Gamma \vDash_{\mathrm{a}} \mathsf{brk}(\ell, v) : E} \text{(SA-BRK)}$$

Fig. 9. Capture Kinding for Labels, Relaxed typing definitions (selected rules) and Answer Typing rules used in the metatheory statements. The full relaxed typing judgments are given in Appendix A.1.

relaxed typing as an invariant during evaluation, before re-normalizing the final result back into standard MNF typing.

Typing Labels. The rule (RT-LABEL) allows typing a label value present in the label context Σ at the corresponding Break $[S]$ shape type. The interesting part lies in the capture set of the value: a label has a single classifier k, so it can be projected to any kind ϕ, so long as it contains the classifier (i.e., $k \in \phi$). This is similarly presented in the additional capture kinding rules for labels, where (K-LABEL) allows kinding a (valid projection of a) label at any kind containing its classifier, while (K-LABEL-ABSURD) sees a projection outside of the label's classifier as the empty set. This allows us to freely change projections of a label within a capture set, as long as they remain valid, while retaining the accessibility of the label itself during computation of runtime labels.

Non-MNF Terms. Normal typing rules easily transfer to non-MNF terms: note that judgments such as $D; \Gamma \vdash x : S \wedge C$ for variables can always be stated more precisely as $\{x\}; \Gamma \vdash x : S \wedge C$. We can always relax such premises into $C; \Gamma \vdash t : S \wedge C$, knowing that when x shows up in the use set, its value is being *used*, and so its *potential use set* (i.e., its annotated capture set during static typing) will be charged. There are, however, two exceptions requiring careful consideration: term application (APP) and packing (PACK). For application, we are more flexible about which capture set gets substituted into the dependent result type: while the original typing implicitly widens the argument variable's type to T and substitutes $\{y\}$ for z, the relaxed rule makes the widening step explicit (RT-APP). For packing, generalizing to terms risks making the calculus no longer call-by-value, so distinct rules for variable (RT-PACK-VAR) and value packing (RT-PACK-VAL) are introduced.

4.3 Main Results

The central result is safety of checked evaluation. Given a well-typed term under the top-level (i.e., empty) context and a label allowance of at least the runtime labels of the use set, evaluation never gets stuck and produces a well-typed answer in the result label context.

Theorem 4.1 (Safety of Checked Evaluation). *If $C; \Sigma; \emptyset \vdash t : E$ and $C^*_\Sigma \subseteq F$ and $\Sigma \vdash t \Downarrow^?_F a$, then evaluation does not get stuck, and $\exists w, \Sigma' \supseteq \Sigma$ such that $a = \langle \Sigma', w \rangle$, and $F; \Sigma'; \emptyset \vDash_a w : E$.*

where $\Sigma \vdash t \Downarrow^?_F a$ means that evaluation terminates with answer a or gets stuck (i.e., it does not diverge). The proof proceeds by induction on the big-step derivation. The full development can be found in Appendix A.4 and is mechanized in Lean 4 (Appendix B.5). Note that in the case of returning a value, we immediately obtain type safety:

Corollary 4.2 (Type Safety). *If $C; \Sigma; \emptyset \vdash t : E$ and $C^*_\Sigma \subseteq F$ and $\Sigma \vdash t \Downarrow_F \langle \Sigma', v \rangle$, then $\emptyset; \Sigma'; \emptyset \vdash_\star v : E$. Equivalently, $\emptyset; \Sigma'; \emptyset \vdash \widehat{v} : E$.*

Furthermore, from the links between checked evaluation's label allowance and answer typing, we can derive additional properties about labels, breaks, and capture sets.

Corollary 4.3 (Effect Safety). *If $\emptyset; \Sigma; \emptyset \vdash t : E$ and $\Sigma \vdash t \Downarrow_\emptyset \langle \Sigma', w \rangle$, then $\exists\, v$ such that $w = v$.*

Corollary 4.4 (Used Label Prediction). *If $C; \Sigma; \emptyset \vdash t : E$ and $\Sigma \vdash t \Downarrow^?_{C^*_\Sigma} a$, then evaluation does not get stuck.*

That is, evaluation of t relies only on access to runtime labels of C. In other words, the typing judgment's use set correctly predicts the set of labels used during evaluation of t.

Corollary 4.5 (Capture Prediction). *If $D; \Sigma; \emptyset \vdash t : S \wedge C$ and $\Sigma \vdash t \Downarrow_{D^*_\Sigma} \langle \Sigma', v \rangle$, then*

$$\Sigma'; \emptyset \vdash \lfloor v \rfloor <: C \qquad \text{and} \qquad \lfloor v \rfloor^*_{\Sigma'} \subseteq C^*_{\Sigma'}$$

Recall that a potential use set $\lfloor v \rfloor$ bounds the captures that a value reaches upon usage. We can predict the potential use set of the answer returned by the evaluation of a term from its static typing judgment. This is similar to the Capture Prediction for Terms lemma in $\mathrm{CC}_{<:\Box}$ [6].

Corollary 4.6 (Handler Coverage). *If $D; \Sigma; \Gamma \vdash t : E$ and $\Sigma \vdash t \Downarrow_{D^*_\Sigma} \langle \Sigma', brk(\ell, v) \rangle$, then:*

(1) Breaks. $\ell \in D^*_{\Sigma'}$.
(2) Intercepts. *If $t = intercept[R, C_t, \phi]$ with h in t', and $(\ell : S :: k) \in \Sigma$, then either h was evaluated, or $k \in \top \setminus \phi$.*

Type-level kind annotations thus predict both which labels may be reached by returned values and how `intercept` partitions propagating breaks into handled and escaping ones.

Two further consequences, found in Appendix A.5, clarify how the control operators behave. Fresh labels introduced by a boundary are local and never appear in the final answer. And for the pass rule of `intercept`, if the handler's own captures exclude labels of the intercepted kind, then no break whose classifier belongs to ϕ can escape.

5 Implementation of Capability Classifiers in Scala

This section describes how the Scala 3 capture checker implements the classifiers of System Capless($\mathcal{K}$) (Section 3), and how they fare in practice.

Projections in Source Code. The `.only` and `.except` projections of Section 2 map directly to the projected captures of Section 3. A chain of projections denotes a single one: `a.only[k1].except[l1]` stands for $(a \mid (k_1 - l_1))$. Source code has no union syntax for kinds. To project to a union of subtrees, the programmer writes a union of projected references:

`{a.only[k1].except[l1], a.only[k2].except[l2]}` $\hat{=}$ $\{(a \mid (k_1 - l_1) \cup (k_2 - l_2))\}$

Every reference then projects to a single subtree, which keeps subkinding simple. Capture subsetting compares a kind against the union of the kinds of all projections of the same reference, by (S-MERGE).

Declaring Classifiers. Since classifiers are the plain traits of Section 2, the classifier tree of Section 3.1 is a slice of Scala's nominal subtype hierarchy, rooted at `Capability`, and the compiler answers sub-classifier queries by subtype tests. Scala traits allow multiple inheritance, so the compiler rejects any class that inherits two unrelated classifier traits, preserving the sub-classifier-or-disjoint property that Section 3.1 relies on.

Representation in the Capture Checker. The compiler retrofits classifiers onto the existing capture checker [59], whose data structures and constraint solver revolve around capture sets of references. Classifiers add one new form of reference, the counterpart of the projected captures π of Figure 3:

```
// x.only[o].except[e1]...except[en], the exclusions strictly below o
case class Classified(underlying: Capability, only: ClassSymbol, except: List[ClassSymbol]) extends DerivedCapability
```

A `Classified` reference represents the projected capture $(x \mid (o - \overline{e}))$. Its `only` and `except` fields form one holed subtree of Section 3.1. Kinds never appear as objects of their own, and solver and set operations carry over unchanged. The cost surfaces in coverage checking: a projection of `b` may be covered by no single reference and still be covered by several projections of `b` together. The checker decides such cases with the remainder computation of Figure 10, which subtracts each covering subtree from the covered one and accepts when nothing remains, deciding subkinding as emptiness of subtraction. Each branch implements one subtraction rule of Figure A.2, named in its comment.

Normal Forms and Subcapturing. A smart constructor maintains the invariants noted in the comments of `Classified`: chained `.only` projections collapse to their least classifier, exclusions outside `only` drop, an exclusion covering `only` empties the whole reference (rules (E-ABSURD) and (S-ABSURD)), and the exclusion list stays pruned and sorted. The checker never meets a degenerate kind. Subcapturing $\Gamma \vdash C_1 <: C_2$ then asks, for each reference in C_1, whether C_2 accounts for it in one of three ways. A single reference subsumes it, following paths and variable bounds, rules (SC-VAR) and (SC-BOUND), and weakening the classifier by (S-SUBKIND). Several projections of the same reference cover it together, decided by the remainder computation of Figure 10, by rule (S-MERGE). Finally, the capture set of its type accounts for it recursively.

Cost of the Kind Algebra. Classifier comparisons are single subtype tests, and most kind operations are polynomial. Membership and emptiness reduce to linear scans, intersection and disjointness to the pairwise intersections of the two kinds' subtrees. The exception is subtraction, and with it subkinding. One call to the remainder computation of Figure 10 costs $O(e)$ subtype tests for a subtrahend with e exclusions and returns at most $e+1$ subtrees, since the `st-sub-r` branch adds one subtree per exclusion it meets. Subtracting a whole kind iterates the computation over its subtrees, so a kind with s subtrees can grow to $O(s\,(e+1)^n)$ subtrees under a subtrahend with n. In practice the exponent stays small: annotation kinds hold one or two subtrees, exclusion lists rarely go beyond a handful of classifiers, and the final `filterNot` of Figure 10 normalizes away empty subtrees. Accordingly, we observe no noticeable compile-time regression against the capture checker of System Capless, whose checking phase takes under 15% of total compilation time [59, Table 1]. A dedicated performance study across compiler and ecosystem is future work.

```
// (o1 - e1) \ (o2 - e2), as a union of subtrees
type Subtree = (Cls, List[Cls])            // o − ē
def subtract(o1: Cls, e1: List[Cls],
             o2: Cls, e2: List[Cls]) =
  def go(ex: List[Cls]): List[Subtree] = ex match
    case Nil => (o1, o2 :: e1) :: Nil      // st-tree
    case l :: ls =>
      if !l.isSub(o2) then go(ls)          // st-irrel-r
      else if l.isSub(o1) then (l,e1)::go(ls) // st-sub-r
      else if o1.isSub(l) then (o1,e1)::Nil   // st-sub-l
      else go(ls)                          // st-irrel-l
  go(e2).filterNot(isEmpty)
```

Fig. 10. The kind remainder computation, abridged. Comments name the subtraction rules of Figure A.2.

Capability Classes. Section 2 tags a class with a classifier through its `extends` clause. We call such classes *capability classes*:

```
class P(val q: Q^) extends Control
```

Every instance of `P` is a capability classified under `Control` and enters capture sets as the projection `any.only[Control]`. The instance's captures must fit within that projection, so a classified class can hold only capabilities under its classifier. The field `q` above must therefore either have a `Control`-classified type `Q`, or constrain its capture set to a `Control` subset, as in `q: Q^{any.only[Control]}`. In exchange, clients may assume the tighter default: the shorthand `P^` in parameter position stands for `P^{any.only[Control]}` rather than `P^{any}`. The declared classifier bounds the capabilities an instance holds rather than tagging them all exactly. Note that the classifier `Control` does not bound a class at *exactly* `Control`, but at the *subtree rooted at* `Control` (as the projection on `any` indicates). This looseness enables encapsulation: a class may keep private members at finer sub-classifiers, and clients can track those subsets (Section 6.3).

Capture Inference. Capture checking runs as its own phase after type checking and local type inference, solving subset constraints over capture set variables on the typed program [60, 59]. An inferred set contains the exact references a term uses, such as the path `file.read`, and never a projection. A projection such as `file.only[Read]` covers a whole subset of the capabilities reachable through `file`, a coarser statement. Writing one is a deliberate widening, which belongs in annotations at API boundaries, in particular where the precise path is hidden (Section 6.3). Projections from annotations flow through the solver's subset constraints like any other reference.

Standard-Library Coverage. By `tokei`'s count of non-comment, non-blank lines in sources compiled under capture checking, the standard library is 81.7% capture-checked (45,146 of 54,992 lines) once classifiers and the accompanying library changes are in place, up from 66.7% (36,605 of 54,841 lines) on the classifier-free Scala 3.8.0 baseline, which already incorporates System Capless. Classifiers do not account for the whole difference, but without them types such as `Try` and `Future` and their packages `scala.util` and `scala.concurrent` cannot be typed precisely at all. The annotation burden is small: typing these signatures precisely takes 14 `.only` and 151 `.except` projections across the whole library.

Impact on Existing Code. Adopting classifiers changes almost no existing capture-checked code. Fully polymorphic capture sets keep their meaning under the top classifier `Capability`, so the capture-checked collections of System Capless need no classifier-specific changes. The one source-level incompatibility concerns capability declarations. Scala seals `Capability`, so a user-defined capability must extend one of its two standard children of Section 2, `SharedCapability` or `ExclusiveCapability`. The split serves Scala's tracking of exclusive capabilities [1], but not classifiers. Sealing the root does not conflict with the open classifier tree of Section 3.1, which gives every classifier unboundedly many children. The checker never treats the two children as exhausting `Capability`: excluding both still leaves a nonempty kind.

6 Discussion

We discuss design decisions surrounding classifiers and their implementation.

6.1 Why a Tree of Classifiers?

A separate kind. Set-based [28] and boolean-algebraic [25, 10] effect systems can express a finite subset of effects, or the effect universe except a finite subset, but not an open specification: for example, `Control` capabilities include not only those defined in the standard library, but also an unbounded number of capabilities in other modules. By separating classifiers from capabilities, we provide a simple solution to classification: capabilities may tag themselves with new or existing classifiers, and the type system treats the set of capabilities in a particular classifier as unbounded.

We deliberately decouple capability classifiers from shape types for simplicity: classifiers have a much more rigid structure than types. This is reflected in System Capless($\mathcal{K}$), where classifier and kind operations are algorithmic and total. Furthermore, this separation allows the core capture calculus to evolve mostly independently, since the integration surface is small (minor changes in subset calculation and one additional subcapturing rule). The Scala implementation also benefits from this restriction: a single-inheritance trait system for classifiers is simple to work with.

The need for hierarchies. A system with flat classifiers (i.e., no sub-classifiers) was also considered. However, classifiers tend to organize into hierarchies, especially permission classifiers such as `Read` and `ReadWrite` (Section 6.3). Flat classifiers would instead require multiple tags on a single capability, as in a tagging system where a read-write reference must carry separate read and write tags, placing a heavy burden on users to tag their capability classes. An open tree structure (or more flexible structures, e.g., lattices) models such hierarchies more naturally.

Disjoint decompositions. However, an open lattice is too flexible: we lose the ability to reason about disjointness. Two sub-lattices upper-bounded by A and B in an open lattice are never disjoint, since a future module can always create a new classifier C directly under A and B. Disjointness is useful in practice: capability classes that split their aggregated capabilities into sub-groups need a way to guarantee that these groups are disjoint. For example, a `Database` capability may partition its usage by table, modeled as sub-classifiers of `Database`. Without disjointness, it

```scala
trait ErrHandler extends Classifier, Control

object RunWithErrHandler extends ErrHandler:
  type Fiber[_] // threaded computation
  type Raise    // exception-raising capability
  def apply[A](
    /* f must not create ErrHandlers */
    f: this.Raise ->{any.except[ErrHandler]} A,
    /* cannot use exceptions */
    handleErrNoRaise: Exception ->
      {any.except[Control]} Fiber[Unit],
  ): Fiber[A]
```

```scala
trait Commit extends Classifier, SharedCapability
trait Rollback extends Classifier, SharedCapability

trait Transaction:
  val commit: () ->{this.only[Commit]} Unit
  val rollback: () ->{this.only[Rollback]} Unit
  /* other operations... */

// runs body with the transaction, commit on end
def withTransaction[T, E^](
  body: (tx: Transaction^) ->
    (() ->{E, tx.except[Commit].except[Rollback]} T)): T
```

Fig. 11. Effect exclusion in real-world APIs: `RunWithErrHandler` from `dune` (left) and `withTransaction` from `google-cloud-go` (right).

is not possible to assert that `db.readUsers` uses `db.only[UserTable]` and not `db.only[TransactionTable]`.

```scala
trait Database extends Classifier, ExclusiveCapability
trait UserTable extends Classifier, Database
/* Similarly, TransactionTable, ProductTable ... */

class DBImpl extends Database:
  val readUsers: () ->{this.only[UserTable]} List[User]
```

```scala
// does not type check with a lattice!
val doesntTouchTransactions :
    () ->{db.except[TransactionTable]}
    List[User]
  = db.readUsers
```

Flix [25, 24] describes disjoint subsets with effect variables or associated effects, but disjointness must be requested explicitly. The Flix analogue of `readUsers` subtracts every sibling table from its effect, as in `List[User] \ {UserTable - TransactionTable - ProductTable}`. Boolean-algebraic subtyping faces the same problem: the effect parameter `UserTable` must be a subtype of the negation of all preceding parameters. InvalML [10] provides the outer scope variable ω to refer to union of visible effect parameters in scope, but are limited to region variables specifically. A classifier tree approach enjoys disjointness by definition.

6.2 Fine-Grained Capability Classification on References

Lutze et al. [25] compile a set of real-world effect exclusion patterns. We succesfully replicated all 59 case studies in Scala with classifiers. In this section, we show how classifiers enable fine-grained control over individual references that Flix's global exclusions cannot express.

Simple effect exclusion. Many patterns simply restrict a closure parameter from performing certain effects. We model this by creating a new classifier for the kind of effects to be disallowed. For instance, from the `amule` example[3], we can model a function that must not invoke any `CSafeIOException`:

```scala
trait CSafeIOException extends Classifier, Control
case class CFileDataIO[T](doRead : (T, Long) -> {any.except[CSafeIOException]} Long)
```

In addition to preserving semantics, we can also mark `CSafeIOException` as part of `Control` capabilities, such that external code may properly reject usage of C exception handlers.

A more involved example comes from `dune`, where the `runWithErrHandler` function requires the body parameter `f` to not nest another `runWithErrHandler` call, and the handler function `handleErrNoRaise` may not raise exceptions. We capture the requirements by creating a classifier `ErrHandler` and giving it to the global capability `RunWithErrHandler` (Figure 11, left).

Here, `f` can throw any exceptions but cannot mention `RunWithErrHandler`, since it is under the `ErrHandler` classifier. Note, however, that the specification is ambiguous on whether `handleErrNoRaise`

```
/* User-defined capability classifiers */
trait ReadWrite extends Classifier, SharedCapability
trait Read extends Classifier, ReadWrite


trait File(val name: String):
  // capability members, ensured to be erased
  erased val r: Read^
  erased val rw: ReadWrite^

  // operations guarded by capability members
  val read: () ->{r} Int
  val write: Int ->{rw} Unit

// runs f with an open file, closes it after use
def withFile[T](f: File^ => T): T
```

(a)

```
class List[+T]:
  def parMap[U](f:T ->{any.only[Read]} U): List[U]

withFile: file =>
  val list = List(99,97,112,121,98,97,114,97)
  list.parMap: item =>
    file.read()    // ok
    file.write(42) // error: capturing ReadWrite
```

(b)

```
trait File(val name: String):
  erased private val r: Read^
  erased private val rw: ReadWrite^
  val read: () ->{this.only[Read]} Int
  val write: Int ->{this.only[ReadWrite]} Unit
```

(c)

Fig. 12. Read-only parallel map: (a) capability declarations, (b) their use, and (c) encapsulating the capability members.

can *use* another error handler in its body. The declaration in Figure 11 disallows it (as `Control` is a parent of `ErrHandler`), but we can allow this by adding `any.only[ErrHandler]` (to allow all error handlers) into `handleErrNoRaise`'s capture set. Going further, we can allow *only* the same `RunWithErrHandler` mechanism to be used, by setting the capture set to `{this, any.except[Control]}`.

Per-reference privilege restriction. Where classifiers go beyond Flix is in restricting specific *references* rather than globally banning effect classes. Another example from Lutze et al. [25] highlights this: the `withTransaction` function from the `google-cloud-go` example[3] takes a closure with a `Transaction` as its input. This closure can use the `Transaction`, but *must not* call `commit` or `rollback` on it. The Flix implementation solves this by completely banning usage of *all* `Commit` and `Rollback` effects, which is too conservative. By precisely projecting the transaction's identity in the capture set, we can do better (Figure 11, right). The `withTransaction` function accepts a closure that takes a `Transaction`, and then *captures* a limited variant of this transaction in its output closure. Note that the parameter closure itself is *pure*: it has no capture set, meaning it does not perform any operations on its own other than pure computation. The *returned* function value however may use `tx`, but not its `Commit` nor `Rollback` subset. Interestingly, by having a capture variable `E` in its captures, it can use other capabilities, including `Commit`/`Rollback` capabilities of *other* transactions. Note that `E` is defined outside of `tx`'s scope, so it cannot contain `tx`.

6.3 Constraining Between Levels of Classifiers

Beyond exclusions on different classifiers, the flexibility of classifiers allows us to express complex restrictions between levels of classifiers. We present an example from Osvald et al. [35], which was also posed as a challenge for capture-checking systems in [55]. In this example, we have a parallel map operation on a `List` data structure that should not perform any write operations, as non-deterministic writes are one of the most common sources of data races. On the other hand, non-deterministic reads are safe in the absence of writes.

We introduce two new classifiers (Figure 12) `Read` and `ReadWrite`, with the former a sub-classifier of the latter. Each `File` now holds two compile-time-only capabilities corresponding to these classifiers, and operations on the file now capture these capabilities. Note that calling these operations will "use" the capability that was captured.

[3]The original definitions can be found in `amule-project/amule` and `google/google-cloud-go` GitHub repositories.

```scala
def logErrors(f: () => Unit): Unit =
  try
    f()
  catch case (exc: Exception)
             (using ct: CanThrow[exc.type]^) =>
    println(exc.toString())
    throw exc // intercept & rethrow
```

```scala
case class Failure(e: Exception, ct: CanThrow[e.type]^)
def captureException(f: () => Unit)
    : Option[Failure^{f.only[Control]}] =
  try
    f(); None
  catch case (exc: Exception)
             (using ct: CanThrow[exc.type]^) =>
    Some(Failure(exc, ct)) // store exc and its CanThrow!
```

Fig. 13. Intercepting handlers: `logErrors` rethrows the caught exception (left), `captureException` stores it together with its `CanThrow` capability (right).

When type-checking the closure passed into `parMap`, the presence of the `file.rw` capability causes the compiler to reject the call, due to the capture set {`file.r`, `file.rw`} not satisfying the capture kind `Read`. On the other hand, without the `file.write` call, the example would compile.

We can go further and encapsulate the compile-time capabilities (Figure 12c), using `this.only[Read]` to refer to the subset of the File itself that contains only `Read` capabilities. The compiler guarantees that the `File` implementation will not capture `rw` in `read`. We however gain a clearer public API, free of the "administrative" capabilities of the implementation. On the other hand, `parMap` should also be more permissive with capabilities that are *neither* reading or writing to references. Depending on API considerations, they can be allowed by including `any.except[ReadWrite]` in `f`'s capture set.

6.4 Non-Lexical Exceptions with Lexical Effect Handlers

Capture checking calculi [6, 59] promote lexical effect handling, where scoped effects like exceptions [33] are assumed to be handled by tunneling [62]. This is, however, not the case for exception handling on the JVM [63] or JavaScript, two main compilation targets for Scala. To maintain compatibility with these exception semantics, we can extend the `CanThrow` model to allow *intercepting* handlers by including the `CanThrow` capability as part of the thrown exception. Any `catch` clause can then rethrow the same exception type as the caught exception, while effect safety is retained.

In Figure 13 (left), `logErrors` prints exceptions thrown by `f` and then rethrows them. Note that apart from those captured by `f`, `logErrors` does not require any additional capabilities: exception safety is retained even when we rethrow the exception, since a handler for it is already known to exist. The `CanThrow` capability has the singleton type of `exc`, because the tested type `Exception` is too coarse: the original exception handler may not handle a different subtype of `Exception`.

One question remains: what if we were to *store* this capability? In Figure 13 (right), `captureException` *stores* the `CanThrow` capability and returns it, allowing the caller to unpack the capability and rethrow the exception. What does the resulting `Failure` capture? Classifiers provide a solution: since `CanThrow` capabilities have the `Control` classifier, we can filter the capture set down to `f.only[Control]`, which contains all possible `CanThrow` capabilities.[4]

6.5 Classifiers as Traits

The classifier universe presented in this paper is distinct from types for pragmatic reasons. However, in the Scala implementation, classifiers are simple traits. While they are restricted, we can still leverage this fact to impose additional constraints on implementing capabilities.

Scala's `Unscoped` classifier is one such example: it represents capabilities that are self-contained and can be returned from a scope, such as owned, mutable data structures. The class restriction requires all `Unscoped` values to capture only other `Unscoped` values. This allows the compiler, with

[4]An astute reader may recognize that this is the same situation as `Try[T]` in Section 2: this is exactly the reasoning behind the capture sets for its constructor.

uniqueness tracking [1], to safely treat returned `Unscoped` values as completely fresh, enabling capabilities to be tracked in a non-scoped way. Formalizing this extension is future work.

As another example, serializable closures require all their captures to also be serializable. Spores [30] track this in Scala by strictly controlling captured variables through a DSL implemented as Scala macros, forcing the user to declare all captures in advance. With classifiers, we can fuse the serializability requirement and functionality into a single classifier trait:

```
trait Serializable extends Classifier, SharedCapability:
  def serialize(): Array[Byte]
  def deserialize(from: Array[Byte])
class Spore(inner: () ->{any.only[Serializable]} Unit)
```

The `Spore` class accepts closures that capture only serializable references. The compiler can then use this information to safely derive serialization methods for such closures, while the usage of spores remains simple and expressive: they are still ordinary function types.

6.6 Limitations

Classifiers rely on subkinding alone, with no kind variables, so the system cannot constrain two unknown capture sets to be disjoint, e.g., a parallel combinator quantifies over two capture variables,

```
def fork[A, B, E1^, E2^](f: () ->{E1} A, g: () ->{E2} B): (A, B)
```

but no bound makes `E1` and `E2` disjoint, so `fork` cannot promise non-interference. Flix [25] states this disjointness with effect differences `p1: T \ {e1 - e2}, p2: U \ {e2 - e1}`. InvalML [10] states it as a negation bound $\beta \leq \neg\alpha$ between quantified effect variables, which transposed to our setting would be a capture-set bound $E_2 \leq \neg E_1$, negating sets of references rather than kinds. Capture checking addresses non-interference through an orthogonal separation mechanism instead, formalized in System Capybara [1] and shipped as Scala's separation checking [47].

The kind algebra checks given kinds and never solves for an unknown one. In Flix, passing an `IO` function to a parameter of effect `ef - Throw` makes boolean unification derive `ef` from `IO = ef - Throw`. Our checking never needs such a step, since a call site requires some instantiation below its bound rather than a most general solution, and capture inference finds the least one. Every exclusion pattern of Section 6.2 type-checks this way. The difference surfaces only at API boundaries, where projections must be written by hand because inference never produces them (Section 5).

Being subtree-based, classifier kinds don't have non-trivial meets: two sibling subtrees always meet at an empty kind. Therefore, a capability cannot be classified at two sibling classifiers: The compiler rejects `class Log extends Read, Serializable` outright (Section 5), and requiring `.only[Read]` and `.only[Serializable]` on a capture set collapses it to empty. Qualifier lattices have exact meets: in System $F^{Q}_{<:}$ [19], variables can be quantified as $l_1 \wedge l_2$ where l_1, l_2 are both base lattice elements (corresponding to a single classifier), and can admit both declarations. Neither algebra subsumes the other: the tree lacks meets, but enjoys complements and disjoint decompositions.

7 Related Work

Capability-based Effects. Type-and-effect systems date back to Lucassen and Gifford's polymorphic effects [23]. Craig et al. [9] show that capabilities can be tracked in a type system to ensure capability safety and reason about effects. Brachthäuser et al. [8] view effect types as capabilities required from the calling context, enabling lightweight effect polymorphism without effect variables, while coeffect systems [36] study contextual requirements more generally. This perspective is close to capture tracking, since a capture set records which capabilities a term requires from its environment. Capture calculus [6, 59, 58] tracks captured capabilities in Scala types, and our calculus builds directly on Xu et al. [59]. Tang and Lindley [49] give a uniform encoding of row-based and capability-based systems into modal effect types.

Capture checking is more than a lightweight effect system. A capture set names program values, often paths such as `body`, `tx`, or `file.r`, so capture checking itself reasons about capabilities by the identity of the values that carry them, keeping two file handles `file1` and `file2` distinct. Classifiers inherit this precision: a projection like `tx.except[Commit]` restricts one particular transaction rather than a whole effect class (Sections 6.2 and 6.3). The same capturing-types discipline reaches further, to sharing, lifetimes, aliasing, and separation of resources [58, 1], which traditional effect systems do not track directly. TACIT builds agent security on the purity of empty capture sets [34].

Effect Classification and Exclusions. Flix [25] introduces effect exclusion on a boolean algebra of effect sets, typing functions that accept any effects *except* a given set, with full Hindley-Milner inference by boolean unification. Checking there is equality-based, ours subsumption-based [51, 53]. Boolean unification keeps the top of the algebra symbolic to stay sound in an open world, and the classifier tree keeps the top of every subtree symbolic (Section 6.1). We replicate Flix's case studies in Section 6.2, and Section 6.6 states what Flix can express that classifiers cannot.

Boolean-algebraic type-and-effect systems in general incorporate intersection and negation into effect typing [27, 10]. They are more expressive than our system because they allow effect set variables in algebraic expressions, but also considerably more complex. Classifiers can also represent exclusion of open sets of classifiers, which does not appear to be representable with intersection and negation of finite sets.

Row typing offers another route to classifying and filtering effects. Rémy's record calculus attaches presence and absence flags to row labels [39], and the Rose language of Morris and McKinna [32] derives absence and disjointness from qualified predicates over rows, so both can state that a named label is absent even from an otherwise open row. Neither calculus, as defined, can state absence of an open category: a row is a finite list of named labels over one uniform tail, so it can mark finitely many labels absent or close the remainder entirely, but cannot exclude a whole open-ended category while leaving the rest open. That is what `any.except[Control]` expresses, excluding the `Control` subtree together with every sub-classifier that later modules add. Rows could recover this by kinding the label universe into categories, which rebuilds the classifier tree on the label side. The modal encoding of Tang and Lindley [49] discussed above suggests a formal bridge between rows and capabilities.

Effect masking [5, 22, 20, 54, 50] provides a way to skip a certain effect handler at runtime when multiple handlers are installed. Biernacki et al. [5] formalize related selective forwarding via a lift operator in a logical-relations semantics of handlers, and Eff [4], Koka [20, 21], Frank [22], and scoped handlers [54] study handlers over open effect rows. Masking and lifting re-route the search for the nearest handler, whereas classifier projections subtract categories from what a reference may reach. Our `intercept` construct is where the two perspectives meet: it lets lexically scoped `boundary`/`break` control temporarily take on nearest-handler semantics, intercepting matching breaks, while the intercepted label's capture set still governs where that label may be retained and used.

First-class names [56] distinguish scoped effect instances by explicit names, a complement to classifiers, which group capabilities into extensible semantic categories with subtree-based filtering.

Xu et al. [57] discuss a subtraction operator and disjointness on types. Their operations are algebraically similar to ours, but serve a type system with records and universal quantification for reasoning about interfaces and overloading rather than capabilities or effects.

Classifying Specific Capability Classes. Spores [30] constrain what may appear in a closure's captured environment, enforcing serialization and concurrency safety. This is close to our `only` and `except` operators, which also restrict which capability values a closure may capture. Spores attach these restrictions to each closure type, whereas our system factors them through a shared, open classifier hierarchy. We further discuss spores and classifier traits in Section 6.5.

Osvald et al. [35] classify values by lifetime scope, and Xhebraj et al. [55] extend this with delayed stack reclamation. Our approach can provide equivalent static guarantees (Section 6.3), though similar stack-allocation optimizations remain future work.

Fractional capabilities and related qualifiers can track ownership and mutability. Haller and Odersky [14] classify references as unique or borrowed. Fractional permissions [7] split read and write access, and Gordon et al. [12] use a mutability lattice. Marshall and Orchard [29] encode fractional ownership with graded modal types. Qualified types [19] likewise attach lattice-based restrictions to values. We do not directly tackle ownership, but classifiers can express closely related distinctions, as discussed in Section 6.3.

Tracking Exceptions in Types. While exceptions have been present in programming languages since at least Goodenough [11], many languages omit them from static types. Java's checked exceptions [13] face exception-polymorphism problems, motivating Scala effect systems [41, 40] and Swift 2's `rethrows` clause [2]. Wu et al. [54] model exceptions with scoped effect handlers. Related accounts appear in Eff [4], Koka [20], and Frank [22]. Zhang et al. [63, 62] argue that intercepted exceptions are hard to reason about with higher-order functions and propose "tunneling" exceptions to statically known handlers. Lexical effect handlers in Effekt [8] and Lexa [26] generalize this idea. Odersky et al.'s safer exceptions [33] track exception handlers with capabilities. Lexical handlers still do not match the non-lexical exception semantics of current Scala backends, and Section 6.4 discusses how our classifier-based account builds on safer exceptions.

Delimited continuations can be encoded on top of existing exception mechanisms [18], and tunneling can itself be implemented over Java exceptions [63]. Together with recent Scala continuation work [31, 37], this suggests a route to capture-safe lexical effect handlers across all backends. In Scala, this matters because any practical account has to span the JVM, Native, and JS backends.

Dynamic Effect Handlers. Plotkin and Pretnar [38] formalize algebraic effect handlers, Kammar et al. [17] develop their programming idioms, and Yoshioka et al. [61] abstract over the choice of effect representation. A general handler dispatches on the operations of one effect, whereas our `intercept` discharges the open family of labels under one classifier. The route to general handlers in Scala runs through the continuation mechanisms discussed above.

8 Conclusion

Capture checking tracks capabilities by identity, but standard library types such as `Try` and `Future` require reasoning about capabilities by *kind*: retaining only control-flow capabilities, or excluding thread-local ones. We introduced capability classifiers, a tree-structured hierarchy of tags with `.only`/`.except` projections on capture sets, giving precise types to constructs that were previously out of reach. The tree structure ensures disjointness between branches while remaining open to new classifiers in any module. We formalized classifiers as an extension of System Capless, with a decidable kind algebra and kind-annotated projections, and proved type safety via a checked big-step semantics, establishing effect safety, capture prediction, and handler coverage, all mechanized in Lean 4. The system is implemented in the Scala 3 compiler. Overall, classifiers make capture checking more expressive by letting the type system capture the effect distinctions that practical code requires, further strengthening the case for programming with effects and capabilities in a mainstream language.

Acknowledgement

This work is supported by the SNSF Advanced Grant TMAG-2_209506/1, "Capabilities for Typing Resources and Effects".

References

[1] Anonymous. 2026. System Capybara: tracking capabilities for separation and freshness. Under submission. (2026) (cit. on pp. 19, 23, 24).

[2] [SW Mod.] Apple Inc., "Error handling," part of Swift 2015Apple Inc. URL: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/errorhandling/, VCS: https://github.com/swiftlang/swift (cit. on p. 25).

[3] Yuyan Bao, Songlin Jia, Guannan Wei, Oliver Bračevac, and Tiark Rompf. 2025. Modeling reachability types with logical relations: Semantic type soundness, termination, effect safety, and equational theory. *Proc. ACM Program. Lang.*, 9, OOPSLA2, 1837–1864. doi:10.1145/3763116 (cit. on p. 14).

[4] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*, 84, 1, 108–123. doi:10.1016/J.JLAMP.2014.02.001 (cit. on pp. 24, 25).

[5] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with care: Relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.*, 2, POPL, 8:1–8:30. doi:10.1145/3158096 (cit. on p. 24).

[6] 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. doi:10.1145/3618003 (cit. on pp. 1, 3, 12, 13, 16, 22, 23, 35).

[7] John Boyland. 2003. Checking interference with fractional permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings* (Lecture Notes in Computer Science). Radhia Cousot, (Ed.) Vol. 2694. Springer, 55–72. doi:10.1007/3-540-44898-5_4 (cit. on p. 25).

[8] 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. doi:10.1145/3428194 (cit. on pp. 23, 25).

[9] Aaron Craig, Alex Potanin, Lindsay Groves, and Jonathan Aldrich. 2018. Capabilities: Effects for free. In *Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12-16, 2018, Proceedings* (Lecture Notes in Computer Science). Jing Sun and Meng Sun, (Eds.) Vol. 11232. Springer, 231–247. doi:10.1007/978-3-030-02450-5_14 (cit. on p. 23).

[10] Cunyuan Gao and Lionel Parreaux. 2025. A lightweight type-and-effect system for invalidation safety: Tracking permanent and temporary invalidation with constraint-based subtype inference. *Proc. ACM Program. Lang.*, 9, OOPSLA2, 2623–2653. doi:10.1145/3763144 (cit. on pp. 2, 19, 20, 23, 24).

[11] John B. Goodenough. 1975. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18, 12, 683–696. doi:10.1145/361227.361230 (cit. on p. 25).

[12] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012.* Gary T. Leavens and Matthew B. Dwyer, (Eds.) ACM, 21–40. doi:10.1145/2384616.2384619 (cit. on p. 25).

[13] James Gosling, William N. Joy, and Guy L. Steele Jr. 1996. *The Java Language Specification.* Addison-Wesley. ISBN: 0-201-63451-1 (cit. on p. 25).

[14] Philipp Haller and Martin Odersky. 2010. Capabilities for uniqueness and borrowing. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings* (Lecture Notes in Computer Science). Theo D'Hondt, (Ed.) Vol. 6183. Springer, 354–378. doi:10.1007/978-3-642-14107-2_17 (cit. on p. 25).

[15] John Hatcliff and Olivier Danvy. 1994. A generic account of continuation-passing styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 458–471. doi:10.1145/174675.178053 (cit. on p. 5).

[16] Young Bae Jun, Hee Sik Kim, and Eun Hwan Roh. 2004. Ideal theory of subtraction algebras. *Sci. Math. Jpn. Online*, e-2004, 397–402. https://www.jams.jp/scm/contents/e-2004-4/2004-37.pdf (cit. on p. 10).

[17] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action! In *ICFP 2013*, 145–158. doi:10.1145/2500365.2500590 (cit. on p. 25).

[18] James Koppel, Gabriel Scherer, and Armando Solar-Lezama. 2018. Capturing the future by replaying the past (functional pearl). *Proc. ACM Program. Lang.*, 2, ICFP, 76:1–76:29. doi:10.1145/3236771 (cit. on p. 25).

[19] Edward Lee, Yaoyu Zhao, Ondřej 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. doi:10.1145/3649832 (cit. on pp. 2, 23, 25).

[20] Daan Leijen. 2014. Koka: programming with row polymorphic effect types. In *MSFP 2014*, 100–126. doi:10.4204/EPTCS.153.8 (cit. on pp. 24, 25).

[21] Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Giuseppe Castagna and Andrew D. Gordon, (Eds.) ACM, 486–499. doi:10.1145/3009837.3009872 (cit. on p. 24).

[22] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Giuseppe Castagna and Andrew D. Gordon, (Eds.) ACM, 500–514. doi:10.1145/3009837.3009897 (cit. on pp. 24, 25).

[23] John M. Lucassen and David K. Gifford. 1988. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*. Jeanne Ferrante and Peter Mager, (Eds.) ACM Press, 47–57. doi:10.1145/73560.73564 (cit. on p. 23).

[24] Matthew Lutze and Magnus Madsen. 2024. Associated effects: Flexible abstractions for effectful programming. *Proc. ACM Program. Lang.*, 8, PLDI, 394–416. doi:10.1145/3656393 (cit. on p. 20).

[25] Matthew Lutze, Magnus Madsen, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2023. With or without you: Programming with effect exclusion. *Proc. ACM Program. Lang.*, 7, ICFP, 448–475. doi:10.1145/3607846 (cit. on pp. 2, 5, 8, 19–21, 23, 24).

[26] Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. 2024. Lexical effect handlers, directly. *Proc. ACM Program. Lang.*, 8, OOPSLA2, 1670–1698. doi:10.1145/3689770 (cit. on p. 25).

[27] Magnus Madsen and Jaco van de Pol. 2020. Polymorphic types and effects with boolean unification. *Proc. ACM Program. Lang.*, 4, OOPSLA, 154:1–154:29. doi:10.1145/3428222 (cit. on p. 24).

[28] Daniel Marino and Todd D. Millstein. 2009. A generic type-and-effect system. In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*. Andrew Kennedy and Amal Ahmed, (Eds.) ACM, 39–50. doi:10.1145/1481861.1481868 (cit. on p. 19).

[29] Danielle Marshall and Dominic Orchard. 2024. Functional ownership through fractional uniqueness. *Proc. ACM Program. Lang.*, 8, OOPSLA1, 1040–1070. doi:10.1145/3649848 (cit. on p. 25).

[30] Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings* (Lecture Notes in Computer Science). Richard E. Jones, (Ed.) Vol. 8586. Springer, 308–333. doi:10.1007/978-3-662-44202-9_13 (cit. on pp. 23, 24).

[31] Guillem Bartrina I Moreno. 2025. *Lexical Delimited Continuations for Scala 3*. Master's thesis. EPFL. https://infoscience.epfl.ch/entities/publication/5b745359-7d14-4553-a3da-8590f573911c (cit. on p. 25). Master's thesis.

[32] J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.*, 3, POPL, 12:1–12:28. doi:10.1145/3290325 (cit. on p. 24).

[33] Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondrej Lhoták. 2021. Safer exceptions for Scala. In *SCALA 2021: Proceedings of the 12th ACM SIGPLAN International Symposium on Scala, Chicago, IL, USA, 17 October 2021*. Julien Richard-Foy and Sébastien Doeraene, (Eds.) ACM, 1–11. doi:10.1145/3486610.3486893 (cit. on pp. 1, 4, 5, 22, 25).

[34] Martin Odersky, Yaoyu Zhao, Yichen Xu, Oliver Bračevac, and Cao Nguyen Pham. 2026. Securing agents with tracked capabilities. In *CAIS*. ACM, 812–838 (cit. on p. 24).

[35] 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 *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. Eelco Visser and Yannis Smaragdakis, (Eds.) ACM, 234–251. doi:10.1145/2983990.2984009 (cit. on pp. 13, 21, 25).

[36] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A calculus of context-dependent computation. In *ICFP 2014*, 123–135. doi:10.1145/2628136.2628160 (cit. on p. 23).

[37] Nguyên Cao Pham and Martin Odersky. 2024. Stack-copying delimited continuations for Scala Native. In *ICOOOLPS*. ACM. doi:10.1145/3679005.3685979 (cit. on p. 25).

[38] Gordon D. Plotkin and Matija Pretnar. 2013. Handling algebraic effects. *Log. Methods Comput. Sci.*, 9, 4. doi:10.2168/LMCS-9(4:23)2013 (cit. on p. 25).

[39] Didier Rémy. 1989. Typechecking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 77–88. doi:10.1145/75277.75284 (cit. on p. 24).

[40] Lukas Rytz. 2014. *A Practical Effect System for Scala*. Ph.D. Dissertation. EPFL, Lausanne, Switzerland. doi:10.5075/epfl-thesis-5935 (cit. on p. 25).

[41] Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight polymorphic effects. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings* (Lecture Notes in Computer Science). James Noble, (Ed.) Vol. 7313. Springer, 258–282. doi:10.1007/978-3-642-31057-7_13 (cit. on p. 25).

[42] [SW Mod.] Scala, "Try API," part of Scala 3 2026EPFL LAMP. URL: https://nightly.scala-lang.org/api/scala/util/Try.html, VCS: https://github.com/scala/scala3 (cit. on pp. 1, 5, 13).

[43] [SW Mod.] Scala, "Capture checker," part of Scala 3 2026EPFL LAMP. URL: https://nightly.scala-lang.org/docs/reference/experimental/capture-checking/index.html, VCS: https://github.com/scala/scala3 (cit. on p. 3).

[44] [SW Mod.] Scala, "CanThrow API," part of Scala 3 2026EPFL LAMP. URL: https://nightly.scala-lang.org/api/scala/CanThrow.html, VCS: https://github.com/scala/scala3 (cit. on pp. 4, 5).

[45] [SW Mod.] Scala, "boundary API," part of Scala 3 2026EPFL LAMP. URL: https://nightly.scala-lang.org/api/scala/util/boundary$.html, VCS: https://github.com/scala/scala3 (cit. on pp. 5, 13).

[46] [SW Mod.] Scala, "Future API," part of Scala 3 2026EPFL LAMP. URL: https://nightly.scala-lang.org/api/scala/concurrent/Future.html, VCS: https://github.com/scala/scala3 (cit. on p. 5).

[47] [SW Mod.] Scala, "Separation checking," part of Scala 3 2026EPFL LAMP. URL: https://nightly.scala-lang.org/docs/reference/experimental/capture-checking/separation-checking.html, VCS: https://github.com/scala/scala3 (cit. on p. 23).

[48] Boris M. Schein. 1992. Difference semigroups. *Communications in Algebra*, 20, 8, 2153–2169. doi:10.1080/00927879208824453 (cit. on p. 10).

[49] Wenhao Tang and Sam Lindley. 2026. Rows and capabilities as modal effects. *Proc. ACM Program. Lang.*, 10, POPL, 923–950. doi:10.1145/3776674 (cit. on p. 24).

[50] Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2025. Modal effect types. *Proc. ACM Program. Lang.*, 9, OOPSLA1, 1130–1157. doi:10.1145/3720476 (cit. on p. 24).

[51] Yan Mei Tang and Pierre Jouvelot. 1995. Effect systems with subtyping. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, USA, June 21-23, 1995.* Neil D. Jones, (Ed.) ACM Press, 45–53. doi:10.1145/215465.215552 (cit. on p. 24).

[52] Peter Thiemann. 2025. What I always wanted to know about second class values. In *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday (OLIVIERFEST'25)*, 117–127. doi:10.1145/3759427.3760373 (cit. on p. 13).

[53] Keith Wansbrough and Simon L. Peyton Jones. 1999. Once upon a polymorphic type. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999.* Andrew W. Appel and Alex Aiken, (Eds.) ACM, 15–28. doi:10.1145/292540.292545 (cit. on p. 24).

[54] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014.* Wouter Swierstra, (Ed.) ACM, 1–12. doi:10.1145/2633357.2633358 (cit. on pp. 24, 25).

[55] 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 *36th European Conference on Object-Oriented Programming, ECOOP 2022, Berlin, Germany, June 6-10, 2022* (LIPIcs). Karim Ali and Jan Vitek, (Eds.) Vol. 222. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:29. doi:10.4230/LIPICS.ECOOP.2022.15 (cit. on pp. 13, 21, 25).

[56] Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. 2022. First-class names for effect handlers. *Proc. ACM Program. Lang.*, 6, OOPSLA2, 30–59. doi:10.1145/3563289 (cit. on p. 24).

[57] Han Xu, Xuejing Huang, and Bruno C. d. S. Oliveira. 2023. Making a type difference: Subtraction on intersection types as generalized record operations. *Proc. ACM Program. Lang.*, 7, POPL, 893–920. doi:10.1145/3571224 (cit. on p. 24).

[58] 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 pp. 23, 24).

[59] 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, 1726–1753. doi:10.1145/3763112 (cit. on pp. 1, 3, 5–7, 10–13, 17, 18, 22–24, 30).

[60] Yichen Xu and Martin Odersky. 2023. Formalizing box inference for capture calculus. (2023). https://arxiv.org/abs/2306.06496 arXiv: 2306.06496 [cs.PL] (cit. on p. 18).

[61] Takuma Yoshioka, Taro Sekiyama, and Atsushi Igarashi. 2024. Abstracting effect systems for algebraic effect handlers. *Proc. ACM Program. Lang.*, 8, ICFP, 455–484. doi:10.1145/3674641 (cit. on p. 25).

[62] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.*, 3, POPL, 5:1–5:29. doi:10.1145/3290318 (cit. on pp. 22, 25).

[63] Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. 2016. Accepting blame for safe tunneled exceptions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016.* Chandra Krintz and Emery D. Berger, (Eds.) ACM, 281–295. doi:10.1145/2908080.2908086 (cit. on pp. 22, 25).

A The Formal System in Full

This appendix completes the formal development of Section 3 and Section 4. The main text presents System Capless($\mathcal{K}$) in monadic normal form (MNF), where the operands of applications and the bodies of packs are variables (Figures 3, 4 and 7). For the big-step safety proof we work instead with the *relaxed* type system, a proof device that drops the variable restriction on operands and pack bodies so that terms are in direct style. Direct style is the more standard setting for a substitution-based big-step proof. The evaluation rules substitute evaluated values into term bodies, and admitting arbitrary operands and pack bodies lets a single typing judgment classify every intermediate configuration without renormalizing to MNF after each substitution. The relaxed judgment $C; \Sigma; \Gamma \vdash_\star t : E$ appears in Figure 9. We prove safety for the relaxed system and transport it back to MNF.

The development is structured as a round trip.

(1) Every MNF-typed term is relaxed-typed (Lemma A.1).
(2) The checked big-step semantics evaluates relaxed-typed terms, and safety holds for the relaxed system (Theorem A.5).
(3) Any relaxed-typed answer can be normalized back into MNF without changing its type (Lemma A.4), and on MNF terms the two systems agree (Lemma A.2).

Chaining the three steps yields safety for the MNF system of the main text (Theorem A.6). Below we give the relaxed syntax and the full relaxed typing rules (Appendix A.1), the correspondence between the two systems (Appendix A.2), the complete checked big-step evaluation rules (Appendix A.3), and the safety theorems with their corollaries (Appendices A.4 and A.5). All results are mechanized in Lean 4, and Appendix B relates the pencil and paper formulation to the mechanization.

For reference, Figure A.1 reproduces the complete MNF type system of System Capless($\mathcal{K}$), of which Figure 4 in the main text shows only a representative selection. Likewise, Figure A.2 reproduces the complete kind algebra condensed in Figure 5.

A.1 Relaxed Terms and the Relaxed Type System

Figure A.3 defines the relaxed language and the complete relaxed type system. Relaxed terms extend the source syntax of Figure 3 in two ways. First, runtime labels ℓ appear as terms, values and captures, exactly as in the semantic domains of Figure 6. Second, the operand positions of applications, type applications and capture applications, as well as the bodies of packs, may hold arbitrary terms rather than variables. The typing judgment $C; \Sigma; \Gamma \vdash_\star t : E$ carries the label context Σ in addition to Γ, since labels are created only at runtime and their types and classifiers are recorded in Σ. The subtyping, subcapturing and kinding judgments are those of the main text, used with the extended capture syntax and, where labels occur, with the label context. We omit the wellformedness premises of the corresponding MNF rules, they carry over unchanged.

Most rules are the evident generalizations of their MNF counterparts, replacing variable operands by typed sub-terms. Three points deserve attention.

Use sets and capture sets align. In (A-RT-APP), (RT-TAPP) and (RT-CAPP) the function sub-term is typed with a use set that coincides with the capture set of its function type. For a variable operand this is derivable by (RT-VAR) and (RT-SUB), so the MNF rules are special cases. For an arbitrary operand the alignment matters: the same set both types the evaluation of the operand and bounds the labels charged when the resulting closure is entered, which is exactly the invariant that checked evaluation maintains (Appendix A.3).

Two rules for pack. The relaxed system types pack $\langle D, t\rangle$ by two rules. (A-RT-PACK-VAR) admits a variable body, matching the source-level rule (PACK) of Figure 4. (A-RT-PACK-VAL) admits an

Capture Kinding $\boxed{\Gamma \vdash C : \phi}$

$$\frac{x : S\,\hat{}\,C \in \Gamma \qquad \Gamma \vdash C.\mathsf{proj}[\phi_1] : \phi_2}{\Gamma \vdash \{(x \mid \phi_1)\} : \phi_2} \quad \text{(K-VAR)}$$

$$\frac{\Gamma \vdash C : \phi \qquad \phi \preceq \phi'}{\Gamma \vdash C : \phi'} \quad \text{(K-SUB)}$$

$$\frac{\Gamma \vdash C : \phi \qquad \phi\ \textbf{empty}}{\Gamma \vdash C : \phi'} \quad \text{(K-ABSURD)}$$

$$\frac{c <: C \in \Gamma \qquad \Gamma \vdash C.\mathsf{proj}[\phi_1] : \phi_2}{\Gamma \vdash \{(c \mid \phi_1)\} : \phi_2} \quad \text{(K-CVAR)}$$

$$\frac{\Gamma \vdash C_1 : \phi \qquad \Gamma \vdash C_2 : \phi}{\Gamma \vdash C_1 \cup C_2 : \phi} \quad \text{(K-UNION)}$$

$$\frac{c : \phi_1 \in \Gamma}{\Gamma \vdash \{(c \mid \phi_2)\} : \phi_1 \wedge \phi_2} \quad \text{(K-CBOUND)}$$

$$\Gamma \vdash \{\} : \phi \quad \text{(K-EMPTY)}$$

Capture Subsetting $\boxed{C_1 \sqsubseteq C_2}$

$$\frac{C_1 \subseteq C_2}{C_1 \sqsubseteq C_2} \quad \text{(S-ELEM)}$$

$$\frac{\phi\ \textbf{empty}}{\{(\theta \mid \phi)\} \sqsubseteq \{\}} \quad \text{(S-ABSURD)}$$

$$\frac{\phi_1 \preceq \phi_2}{\{(\theta \mid \phi_1)\} \sqsubseteq \{(\theta \mid \phi_2)\}} \quad \text{(S-SUBKIND)}$$

$$\{(\theta \mid (\phi_1 \vee \phi_2))\} \sqsubseteq \{(\theta \mid \phi_1), (\theta \mid \phi_2)\} \quad \text{(S-MERGE)}$$

Subcapturing $\boxed{\Gamma \vdash C_1 <: C_2}$

$$\frac{\Gamma \vdash C_1 <: C_2 \qquad \Gamma \vdash C_2 <: C_3}{\Gamma \vdash C_1 <: C_3} \quad \text{(SC-TRANS)}$$

$$\frac{C_1 \ \boxed{\sqsubseteq}\ C_2}{\Gamma \vdash C_1 <: C_2} \quad \text{(SC-ELEM)}$$

$$\frac{x : S\,\hat{}\,C \in \Gamma}{\Gamma \vdash \boxed{\{}(x \ \boxed{\mid \phi}\ )\boxed{\}} <: C\ \boxed{.\mathsf{proj}[\phi]}} \quad \text{(SC-VAR)}$$

$$\frac{\Gamma \vdash C_1 <: C \qquad \Gamma \vdash C_2 <: C}{\Gamma \vdash C_1 \cup C_2 <: C} \quad \text{(SC-SET)}$$

$$\frac{c <: C \in \Gamma}{\Gamma \vdash \boxed{\{}(c \ \boxed{\mid \phi}\ )\boxed{\}} <: C\ \boxed{.\mathsf{proj}[\phi]}} \quad \text{(SC-BOUND)}$$

$$\frac{\Gamma \vdash C : \phi}{\Gamma \vdash C <: C.\mathsf{proj}[\phi]} \quad (\boxed{\text{SC-PROJ}})$$

Bound Subtyping $\boxed{\Gamma \vdash B_1 <:_{\mathrm{B}} B_2}$

$$\frac{\Gamma \vdash C_1 <: C_2}{\Gamma \vdash C_1 <:_{\mathrm{B}} C_2} \quad \text{(B-SET)} \qquad \frac{\phi_1 \preceq \phi_2}{\Gamma \vdash \phi_1 <:_{\mathrm{B}} \phi_2} \quad (\boxed{\text{B-KIND}}) \qquad \frac{\Gamma \vdash C : \phi}{\Gamma \vdash C <:_{\mathrm{B}} \phi} \quad (\boxed{\text{B-SET-KIND}})$$

Typing $\boxed{C; \Gamma \vdash t : E}$

$$\frac{x : S\,\hat{}\,C \in \Gamma}{\{x\}; \Gamma \vdash x : S\,\hat{}\,\{x\}} \quad \text{(VAR)}$$

$$\frac{C \uplus \{x\}; (\Gamma, x : T) \vdash t : E \qquad \Gamma \vdash T\ \textbf{wf}}{\{\}; \Gamma \vdash \lambda^{C}(x : T)t : (\forall (x : T)E)\,\hat{}\,C} \quad \text{(ABS)}$$

$$\frac{C; (\Gamma, c : B) \vdash t : E \qquad \Gamma \vdash B\ \textbf{wf}}{\{\}; \Gamma \vdash \lambda^{C}[c : B]t : (\forall [c : B]E)\,\hat{}\,C} \quad \text{(CABS)}$$

$$\frac{C; \Gamma \vdash x : (\forall [c : B]E)\,\hat{}\,C' \qquad \Gamma \vdash D <:_{\mathrm{B}} B}{C; \Gamma \vdash x[D] : [D/c]E} \quad \text{(CAPP)}$$

$$\frac{C; \Gamma \vdash x : [D/c]T \qquad \boxed{\Gamma \vdash D <:_{\mathrm{B}} B}}{\{\}; \Gamma \vdash \mathsf{pack}\ \langle D, x\rangle : \exists c\ \boxed{: B}\ .T} \quad \text{(PACK)}$$

$$\frac{C'; \Gamma \vdash x : (\forall (z : T)E)\,\hat{}\,C \qquad C'; \Gamma \vdash y : T}{C'; \Gamma \vdash x\ y : [y/z]E} \quad \text{(APP)}$$

$$\frac{C; \Gamma \vdash t : T \qquad C; (\Gamma, x : T) \vdash u : E \qquad \Gamma \vdash C, E\ \textbf{wf}}{C; \Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ u : E} \quad \text{(LET)}$$

$$\frac{C; (\Gamma, X <: S) \vdash t : E \qquad \Gamma \vdash S\ \textbf{wf}}{\{\}; \Gamma \vdash \lambda^{C}[X <: S]t : (\forall [X <: S]E)\,\hat{}\,C} \quad \text{(TABS)}$$

$$\frac{C; \Gamma \vdash t : E \qquad \Gamma \vdash E <: F \qquad \Gamma \vdash C <: C' \qquad \Gamma \vdash C', F\ \textbf{wf}}{C'; \Gamma \vdash t : F} \quad \text{(SUB)}$$

$$\frac{C'; \Gamma \vdash x : (\forall [X <: S]E)\,\hat{}\,C}{C'; \Gamma \vdash x[S] : [S/X]E} \quad \text{(TAPP)}$$

$$\frac{C; \Gamma \vdash t : \exists c\ \boxed{: B}\ .T \qquad \Gamma \vdash C, F\ \textbf{wf} \qquad C; (\Gamma,\ \boxed{c : B}\ , x : T) \vdash u : F}{C; \Gamma \vdash \mathsf{let}\ \langle c, x\rangle = t\ \mathsf{in}\ u : E} \quad \text{(LET-E)}$$

Subtyping $\boxed{\Gamma \vdash E_1 <: E_2}$

$$\Gamma \vdash S <: \top \quad \text{(TOP)} \qquad \Gamma \vdash E <: E \quad \text{(REFL)}$$

$$\frac{\Gamma \vdash E_1 <: E_2 \qquad \Gamma \vdash E_2 <: E_3}{\Gamma \vdash E_1 <: E_3} \quad \text{(TRANS)}$$

$$\frac{X <: S \in \Gamma}{\Gamma \vdash X <: S} \quad \text{(TVAR)}$$

$$\frac{\Gamma \vdash S_1 <: S_2 \qquad \Gamma \vdash C_1 <: C_2}{\Gamma \vdash S_1\,\hat{}\,C_1 <: S_2\,\hat{}\,C_2} \quad \text{(CAPT)}$$

$$\frac{(\Gamma, c <:\ \boxed{B_1}\ ) \vdash T_1 <: T_2 \qquad \boxed{\Gamma \vdash B_1 <:_{\mathrm{B}} B_2}}{\Gamma \vdash \exists c\ \boxed{: B_1}\ .T_1 <: \exists c\ \boxed{: B_2}\ .T_2} \quad \text{(EXIST)}$$

$$\frac{(\Gamma, X <: S_2) \vdash E_1 <: E_2 \qquad \Gamma \vdash S_2 <: S_1}{\Gamma \vdash \forall (X <: S_1)E_1 <: \forall (X <: S_2)E_2} \quad \text{(TFUN)}$$

$$\frac{\Gamma \vdash E_1 <: E_2 \qquad \Gamma \vdash T_2 <: T_1}{\Gamma \vdash \forall (x : T_1)E_1 <: \forall (x : T_2)E_2} \quad \text{(FUN)}$$

$$\frac{(\Gamma, c : B_2) \vdash E_1 <: E_2 \qquad \Gamma \vdash B_2 <:_{\mathrm{B}} B_1}{\Gamma \vdash \forall (c : B_1)E_1 <: \forall (c : B_2)E_2} \quad \text{(CFUN)}$$

Fig. A.1. Type system of System Capless($\mathcal{K}$) in full (copy of the condensed Figure 4). Changes compared to System Capless [59] are marked .

Intersection $\boxed{\phi_1 \wedge \phi_2 = \phi}$

$$\frac{k_1 \sqsubseteq k_2}{(k_1 - \overline{l_1}) \wedge (k_2 - \overline{l_2}) = k_1 - (\overline{l_1}, \overline{l_2})} \quad \text{(I-SUBTREE-L)}$$

$$\frac{k_2 \sqsubseteq k_1}{(k_1 - \overline{l_1}) \wedge (k_2 - \overline{l_2}) = k_2 - (\overline{l_1}, \overline{l_2})} \quad \text{(I-SUBTREE-R)}$$

$$\frac{k_1 \perp k_2}{(k_1 - \overline{l_1}) \wedge (k_2 - \overline{l_2}) = \emptyset} \quad \text{(I-DISJOINT)}$$

$$\phi \wedge \emptyset = \emptyset \quad \text{(I-EMPTY-R)}$$

$$\frac{\phi \wedge \phi_1 = R_1 \qquad \phi \wedge \phi_2 = R_2}{\phi \wedge (\phi_1 \vee \phi_2) = R_1 \vee R_2} \quad \text{(I-UNION-R)}$$

$$\emptyset \wedge (k - \overline{k_i}) = \emptyset \quad \text{(I-EMPTY-L)}$$

$$\frac{\phi_1 \wedge (k - \overline{k_i}) = R_1 \qquad \phi_2 \wedge (k - \overline{k_i}) = R_2}{(\phi_1 \vee \phi_2) \wedge (k - \overline{k_i}) = R_1 \vee R_2} \quad \text{(I-UNION-L)}$$

Subtraction $\boxed{\phi_1 \setminus \phi_2 = \phi}$

$$(k_1 - \overline{l_1}) \setminus (k_2 - \epsilon) = k_1 - (k_2, \overline{l_1}) \quad \text{(ST-TREE)}$$

$$\frac{k_2 \sqsubset l}{(k_1 - \overline{l_1}) \setminus (k_2 - (l, \overline{l_2})) = k_1 - \overline{l_1}} \quad \text{(ST-ABSURD-R)}$$

$$\frac{k_2 \perp l \qquad (k_1 - \overline{l_1}) \setminus (k_2 - \overline{l_2}) = \phi}{(k_1 - \overline{l_1}) \setminus (k_2 - (l, \overline{l_2})) = \phi} \quad \text{(ST-IRREL-R)}$$

$$\frac{l \sqsubseteq k_2 \qquad l \sqsubseteq k_1 \qquad (k_1 - \overline{l_1}) \setminus (k_2 - \overline{l_2}) = \phi}{(k_1 - \overline{l_1}) \setminus (k_2 - (l, \overline{l_2})) = (l - \overline{l_1}) \vee \phi} \quad \text{(ST-SUB-R)}$$

$$\frac{l \sqsubseteq k_2 \qquad k_1 \sqsubset l}{(k_1 - \overline{l_1}) \setminus (k_2 - (l, \overline{l_2})) = (k_1 - \overline{l_1})} \quad \text{(ST-SUB-L)}$$

$$\frac{l \sqsubseteq k_2 \qquad k_1 \perp l \qquad (k_1 - \overline{l_1}) \setminus (k_2 - (\overline{l_2})) = \phi}{(k_1 - \overline{l_1}) \setminus (k_2 - (l, \overline{l_2})) = \phi} \quad \text{(ST-IRREL-L)}$$

$$\phi \setminus \emptyset = \phi \quad \text{(ST-EMPTY-R)}$$

$$\frac{\phi \setminus \phi_1 = R_1 \qquad R_1 \setminus \phi_2 = R_2}{\phi \setminus (\phi_1 \vee \phi_2) = R_2} \quad \text{(ST-UNION-R)}$$

$$\emptyset \setminus (k - \overline{k_i}) = \emptyset \quad \text{(ST-EMPTY-L)}$$

$$\frac{\phi_1 \setminus (k - \overline{k_i}) = R_1 \qquad \phi_2 \setminus (k - \overline{k_i}) = R_2}{(\phi_1 \vee \phi_2) \setminus (k - \overline{k_i}) = R_1 \vee R_2} \quad \text{(ST-UNION-L)}$$

Emptiness $\boxed{\phi \textbf{ empty}}$

$$\emptyset \textbf{ empty} \quad \text{(E-EMPTY)}$$

$$\frac{k \sqsubseteq l_i}{k - (l_1, \ldots, l_i, \ldots, l_n) \textbf{ empty}} \quad \text{(E-ABSURD)}$$

$$\frac{\phi_1 \textbf{ empty} \qquad \phi_2 \textbf{ empty}}{(\phi_1 \vee \phi_2) \textbf{ empty}} \quad \text{(E-UNION)}$$

Disjoint $\boxed{\phi_1 \perp \phi_2}$ Defined as $(\phi_1 \wedge \phi_2)$ **empty**.

Subkinding $\boxed{\phi_1 \preceq \phi_2}$ Defined as $(\phi_1 \setminus \phi_2)$ **empty**.

Inclusion $\boxed{k \in \phi}$

$$\frac{k \sqsubseteq k_0}{k \in (k_0 - \epsilon)} \quad \text{(IN-BASE)}$$

$$\frac{k \not\sqsubseteq k_1 \qquad k \in (k_0 - (k_2, \ldots, k_n))}{k \in (k_0 - (k_1, \ldots, k_n))} \quad \text{(IN-NONEXCL)}$$

$$\frac{k \in \phi_1}{k \in (\phi_1 \vee \phi_2)} \quad \text{(IN-UNION-L)}$$

$$\frac{k \in \phi_2}{k \in (\phi_1 \vee \phi_2)} \quad \text{(IN-UNION-R)}$$

Capture Set Projection $C.\text{proj}[\phi] := \left\{\overline{(\theta_i \mid \phi_i \wedge \phi)}\right\}$ where $C = \left\{\overline{(\theta_i \mid \phi_i)}\right\}$

Exclusive Union $C \uplus \{x_1, \ldots, x_n\} = C \cup \{(x_1 \mid \top), \ldots, (x_n \mid \top)\}$ where $\forall i, \phi.(x_i \mid \phi) \notin C$

Fig. A.2. Kind algebra of System Capless($\mathcal{K}$) in full (copy of the condensed Figure 5).

arbitrary value body. The distinction exists because evaluation never reduces under a pack. Rule (E-EX-V) substitutes the packed term directly into the continuation of the consuming existential let. Evaluating the body first could widen its use set and thereby break the precise existential

Relaxed Syntax

$s,\ t,\ u \coloneqq$		**Term**
	$x \mid v$	variable, value
	$t\ s$	application
	$t[S]$	type application
	$t[C]$	capture application
	$\mathsf{let}\ x = t\ \mathsf{in}\ u$	let
	$\mathsf{let}\ \langle c, x\rangle = t\ \mathsf{in}\ u$	existential let
$v \coloneqq$		**Value**
	$\lambda^{C}(x{:}T)t \mid \lambda^{C}[X{<:}S]t \mid \lambda^{C}[c{:}B]t$	functions
	$\mathsf{pack}\ \langle C, t\rangle$	pack
	ℓ	label

Typing (values and variables) $\boxed{C; \Sigma; \Gamma \vdash_\star t : E}$

$$\frac{x : S\,^\wedge C \in \Gamma}{\{x\}; \Sigma; \Gamma \vdash_\star x : S\,^\wedge\{x\}} \quad \text{(RT-VAR)}$$

$$\frac{C; \Sigma; \Gamma \vdash_\star t : E \qquad \Sigma; \Gamma \vdash E <: F \qquad \Sigma; \Gamma \vdash C <: C'}{C'; \Sigma; \Gamma \vdash_\star t : F} \quad \text{(RT-SUB)}$$

$$\frac{(\ell : S :: k) \in \Sigma \qquad k \in \phi}{\emptyset; \Sigma; \Gamma \vdash_\star \ell : \mathsf{Break}\,[S]\,^\wedge\{(\ell \mid \phi)\}} \quad \text{(RT-LABEL)}$$

$$\frac{C \uplus \{x\}; \Sigma; (\Gamma, x : T) \vdash_\star t : E}{\{\}; \Sigma; \Gamma \vdash_\star \lambda^{C}(x : T)t : (\forall(x : T)E)\,^\wedge C} \quad \text{(RT-ABS)}$$

$$\frac{C; \Sigma; \Gamma \vdash_\star t : \exists c : B.\, T \qquad C; \Sigma; (\Gamma, c : B, x : T) \vdash_\star u : F}{C; \Sigma; \Gamma \vdash_\star \mathsf{let}\ \langle c, x\rangle = t\ \mathsf{in}\ u : F} \quad \text{(RT-EX)}$$

$$\frac{C; \Sigma; (\Gamma, X <: S) \vdash_\star t : E}{\{\}; \Sigma; \Gamma \vdash_\star \lambda^{C}[X <: S]t : (\forall[X <: S]E)\,^\wedge C} \quad \text{(RT-TABS)}$$

$$\frac{C'; \Sigma; \Gamma \vdash_\star x : [D/c]T \qquad \Sigma; \Gamma \vdash D <:_{\mathsf{B}} B}{\emptyset; \Sigma; \Gamma \vdash_\star \mathsf{pack}\ \langle D, x\rangle : \exists c : B.\, T} \quad \text{(RT-PACK-VAR)}$$

$$\frac{C; \Sigma; (\Gamma, c : B) \vdash_\star t : E}{\{\}; \Sigma; \Gamma \vdash_\star \lambda^{C}[c : B]t : (\forall[c : B]E)\,^\wedge C} \quad \text{(RT-CABS)}$$

$$\frac{\emptyset; \Sigma; \Gamma \vdash_\star v : [D/c]T \qquad \Sigma; \Gamma \vdash D <:_{\mathsf{B}} B}{\emptyset; \Sigma; \Gamma \vdash_\star \mathsf{pack}\ \langle D, v\rangle : \exists c : B.\, T} \quad \text{(RT-PACK-VAL)}$$

$$\frac{C_1; \Sigma; \Gamma \vdash_\star t_1 : (\forall(z : S\,^\wedge C_0)E)\,^\wedge C_1 \qquad C_2; \Sigma; \Gamma \vdash_\star t_2 : S\,^\wedge C_2 \qquad \Sigma; \Gamma \vdash C_2 <: C_0}{C_1 \cup C_2; \Sigma; \Gamma \vdash_\star t_1\ t_2 : [C_2/z]E} \quad \text{(RT-APP)}$$

$$\frac{C; \Sigma; \Gamma \vdash_\star t : \mathsf{Break}\,[S]\,^\wedge C \qquad C; \Sigma; \Gamma \vdash_\star s : S}{C; \Sigma; \Gamma \vdash_\star t\ s : R} \quad \text{(RT-BREAK)}$$

$$\frac{C \uplus \{c, x\}; \Sigma; (\Gamma, c : k, x : \mathsf{Break}\,[S]\,^\wedge\{c\}) \vdash_\star t : S}{C; \Sigma; \Gamma \vdash_\star \mathsf{boundary}[S, k]\ \mathsf{as}\ \langle c, x\rangle\ \mathsf{in}\ t : S} \quad \text{(RT-BND)}$$

$$\frac{C; \Sigma; \Gamma \vdash_\star t : (\forall[X <: S]E)\,^\wedge C \qquad \Sigma; \Gamma \vdash R <: S}{C; \Sigma; \Gamma \vdash_\star t[R] : [R/X]E} \quad \text{(RT-TAPP)}$$

$$\frac{C_t; \Sigma; \Gamma \vdash_\star t : E \qquad C_h; \Sigma; \Gamma \vdash_\star h : \mathsf{Handler}_{\mathsf{pass}}[E, \phi, C_h, C_t]}{C_t.\mathsf{proj}[\top \setminus \phi] \cup C_h; \Sigma; \Gamma \vdash_\star \mathsf{intercept}[E, C_t, \phi]\ \mathsf{with}\ h\ \mathsf{in}\ t : E} \quad \text{(RT-ICP-PASS)}$$

$$\frac{C; \Sigma; \Gamma \vdash_\star t : (\forall[c : B]E)\,^\wedge C \qquad \Sigma; \Gamma \vdash D <:_{\mathsf{B}} B}{C; \Sigma; \Gamma \vdash_\star t[D] : [D/c]E} \quad \text{(RT-CAPP)}$$

$$\frac{C_t; \Sigma; \Gamma \vdash_\star t : E \qquad C_h; \Sigma; \Gamma \vdash_\star h : \mathsf{Handler}_{\mathsf{gen}}[E, \phi, C_h, C_t]}{C_t \cup C_h; \Sigma; \Gamma \vdash_\star \mathsf{intercept}[E, C_t, \phi]\ \mathsf{with}\ h\ \mathsf{in}\ t : E} \quad \text{(RT-ICP-GEN)}$$

$$\frac{C; \Sigma; \Gamma \vdash_\star t : T \qquad C; \Sigma; (\Gamma, x : T) \vdash_\star u : E}{C; \Sigma; \Gamma \vdash_\star \mathsf{let}\ x = t\ \mathsf{in}\ u : E} \quad \text{(RT-LET)}$$

Fig. A.3. Relaxed terms and the full relaxed type system. Subtyping $\Gamma \vdash E_1 <: E_2$, subcapturing $\Gamma \vdash C_1 <: C_2$, capture kinding $\Gamma \vdash C : \phi$ and bound subtyping $\Gamma \vdash B_1 <:_{\mathsf{B}} B_2$ are as in Figures A.1 and A.2, read with the label context Σ and the label rules (K-LABEL), (K-LABEL-ABSURD) of Figure 9 where labels occur. The rules (A-RT-APP), (A-RT-PACK-VAR) and (A-RT-PACK-VAL) repeat Figure 9 for completeness.

witness recorded by the pack, so instead typing insists that the body is already a variable or a value. Substitution maps variables to values, so this invariant is stable under evaluation.

Breaks and labels. (RT-LABEL) types a runtime label at any kind ϕ that contains the classifier recorded in Σ. Labels arise only during evaluation, so the rule has no counterpart in the main-text figures. The kind annotation is chosen by the derivation and need not be the singleton of the recorded classifier. This flexibility is what lets a label allocated inside an intercepted body satisfy the handler's capture requirements. (RT-BREAK) generalizes (T-BREAK): a break invocation never returns, so it can be assigned any wellformed result type.

A.2 Relating the MNF and Relaxed Systems

We write $C; \Sigma; \Gamma \vdash t : E$ for the MNF typing judgment of the main text (Figures 4 and 7), extended pointwise with the label context in the same way as the relaxed system. A relaxed term is *in MNF* if every operand of an application, type application or capture application, and every pack body, is a variable. Source terms are in MNF by construction.

Lemma A.1 (MNF typing embeds into relaxed typing). *If* $C; \Sigma; \Gamma \vdash t : E$ *then* $C; \Sigma; \Gamma \vdash_\star t : E$.

Lemma A.2 (Relaxed typing restricts to MNF typing). *If* $C; \Sigma; \Gamma \vdash_\star t : E$ *and* t *is in MNF, then* $C; \Sigma; \Gamma \vdash t : E$.

For MNF terms the two systems therefore coincide. To transport arbitrary relaxed terms back into MNF we use a standard normalization function.

Definition A.3 (MNF normalization). The function $\widehat{t}$ let-binds every operand that is not already a variable and is a homomorphism on all remaining term forms. The characteristic equations are

$$\begin{aligned}
\widehat{t_1\ t_2} &= \mathsf{let}\ x = \widehat{t_1}\ \mathsf{in}\ \mathsf{let}\ y = \widehat{t_2}\ \mathsf{in}\ x\ y \\
\widehat{t[S]} &= \mathsf{let}\ x = \widehat{t}\ \mathsf{in}\ x[S] \\
\widehat{t[C]} &= \mathsf{let}\ x = \widehat{t}\ \mathsf{in}\ x[C] \\
\widehat{\mathsf{pack}\ \langle D, t\rangle} &= \mathsf{let}\ x = \widehat{t}\ \mathsf{in}\ \mathsf{pack}\ \langle D, x\rangle
\end{aligned}$$

with x, y fresh, where the extra binding is omitted whenever the operand already is a variable.

Lemma A.4 (Normalization preserves relaxed typing). $\widehat{t}$ *is in MNF, and if* $C; \Sigma; \Gamma \vdash_\star t : E$ *then* $C; \Sigma; \Gamma \vdash_\star \widehat{t} : E$.

Combining Lemma A.4 with Lemma A.2 shows that every relaxed-typed term normalizes to an MNF-typed term of the same type and use set.

A.3 Checked Big-Step Evaluation in Full

Figure A.4 lists the complete core rules of the checked big-step semantics, including the break propagation rules that Figure 6 elides. A propagating break simply short-circuits the surrounding evaluation context. Together with the boundary and intercept rules of Figure 8, these are all evaluation rules of the system.

Where the checks fire. The green side conditions perform all access checking. (A-E-APP) checks that the runtime labels of the closure and of the argument are within the current allowance and then evaluates the body under the tightened allowance $D^*_{\Sigma''} \cup v^*_{\Sigma''}$. (A-E-TAPP) and (A-E-CAPP) check the closure's runtime labels only, type and capture arguments carry no labels of their own. (A-E-BREAK) checks that the invoked label is permitted, and (E-ICP-P) rechecks a passing break against the outer allowance. Boundary rules extend the allowance with the fresh label for the extent of the body, and (E-ICP-V), (E-ICP-M) and (E-ICP-P) run the intercepted body under the allowance $(C_t)^*_\Sigma$ induced by its static use set.

Usage is checked, capture is not. No rule inspects the labels captured inside a closure that is merely passed around. A closure may travel through contexts whose allowance excludes its captured labels. The check fires when the closure is entered, at which point its annotation must be within the allowance. This is the operational counterpart of the distinction between capture sets and use sets in the type system.

Big-step Evaluation (complete core rules) $\boxed{\Sigma \vdash t \Downarrow_F a}$

$$\frac{}{\Sigma \vdash v \Downarrow_F \langle \Sigma, v \rangle}\ (\text{E-VAL})$$

$$\frac{\Sigma \vdash t_1 \Downarrow_F \langle \Sigma', v \rangle \quad \Sigma' \vdash [\lfloor v \rfloor / x][v/x]t_2 \Downarrow_F a}{\Sigma \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \Downarrow_F a}\ (\text{E-LET-V})$$

$$\frac{\Sigma \vdash e_1 \Downarrow_F \langle \Sigma', \lambda^D(x:T)t \rangle \quad \Sigma' \vdash e_2 \Downarrow_F \langle \Sigma'', v \rangle \quad D^*_{\Sigma''} \subseteq F \quad v^*_{\Sigma''} \subseteq F \quad \Sigma'' \vdash [\lfloor v \rfloor / x][v/x]t \Downarrow_{D^*_{\Sigma''} \cup v^*_{\Sigma''}} a}{\Sigma \vdash e_1 e_2 \Downarrow_F a}\ (\text{E-APP})$$

$$\frac{\Sigma \vdash t_1 \Downarrow_F \langle \Sigma', \mathsf{pack}\ \langle D, v \rangle \rangle \quad \Sigma' \vdash [D/c][\lfloor v \rfloor / x][v/x]t_2 \Downarrow_F a}{\Sigma \vdash \mathsf{let}\ \langle c, x \rangle = t_1\ \mathsf{in}\ t_2 \Downarrow_F a}\ (\text{E-EX-V})$$

$$\frac{\Sigma \vdash e_1 \Downarrow_F \langle \Sigma', \ell \rangle \quad \Sigma' \vdash e_2 \Downarrow_F \langle \Sigma'', v \rangle \quad l \in F}{\Sigma \vdash e_1 e_2 \Downarrow_F \langle \Sigma'', \mathsf{brk}(\ell, v) \rangle}\ (\text{E-BREAK})$$

$$\frac{\Sigma \vdash e \Downarrow_F \langle \Sigma', \lambda^D[X <: S_0]t \rangle \quad D^*_{\Sigma'} \subseteq F \quad \Sigma' \vdash [S/X]t \Downarrow_{D^*_{\Sigma'}} a}{\Sigma \vdash e[S] \Downarrow_F a}\ (\text{E-TAPP})$$

$$\frac{\Sigma \vdash e \Downarrow_F \langle \Sigma', \lambda^D[c:B]t \rangle \quad D^*_{\Sigma'} \subseteq F \quad \Sigma' \vdash [C/c]t \Downarrow_{D^*_{\Sigma'}} a}{\Sigma \vdash e[C] \Downarrow_F a}\ (\text{E-CAPP})$$

Break Propagation

$$\frac{\Sigma \vdash e_1 \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell, v) \rangle}{\Sigma \vdash e_1 e_2 \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell, v) \rangle}\ (\text{E-APP-B1})$$

$$\frac{\Sigma \vdash e \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell, v) \rangle}{\Sigma \vdash e[C] \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell, v) \rangle}\ (\text{E-CAPP-B})$$

$$\frac{\Sigma \vdash e_1 \Downarrow_F \langle \Sigma', v_1 \rangle \quad \Sigma' \vdash e_2 \Downarrow_F \langle \Sigma'', \mathsf{brk}(\ell, v) \rangle}{\Sigma \vdash e_1 e_2 \Downarrow_F \langle \Sigma'', \mathsf{brk}(\ell, v) \rangle}\ (\text{E-APP-B2})$$

$$\frac{\Sigma \vdash t_1 \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell, v) \rangle}{\Sigma \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell, v) \rangle}\ (\text{E-LET-B})$$

$$\frac{\Sigma \vdash e \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell, v) \rangle}{\Sigma \vdash e[S] \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell, v) \rangle}\ (\text{E-TAPP-B})$$

$$\frac{\Sigma \vdash t_1 \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell, v) \rangle}{\Sigma \vdash \mathsf{let}\ \langle c, x \rangle = t_1\ \mathsf{in}\ t_2 \Downarrow_F \langle \Sigma', \mathsf{brk}(\ell, v) \rangle}\ (\text{E-EX-B})$$

Fig. A.4. Checked big-step evaluation, complete core rules. The first block repeats Figure 6 for completeness, the second block lists the break propagation rules omitted there. The boundary and intercept rules of Figure 8 are already complete.

Unchecked evaluation. Deleting the allowance subscripts and every green side condition of the forms $U \subseteq F$ and $l \in F$ yields an ordinary call-by-value big-step semantics for the relaxed language, written $\Sigma \vdash t \Downarrow a$. The corollaries of Appendix A.5 relate the two judgments: for well-typed terms the checks never fail, so checked and unchecked evaluation agree.

A.4 Safety

Answers are typed by the judgment $F; \Sigma; \Gamma \vDash_a a : E$ of Figure 9. A returned value must be relaxed-typed at E with an empty use set ((SA-RET)). A propagating break $\mathsf{brk}(\ell, v)$ must invoke a label that is bound in Σ and permitted by the allowance F, carrying a payload of the label's answer type ((SA-BRK)).

Throughout, we use the convention that the label context only grows during evaluation, and we identify labels across this growth. The mechanization instead threads explicit injective renamings between label contexts, see Appendix B.7.

Theorem A.5 (Safety of Checked Evaluation). *If $C; \Sigma; \emptyset \vdash_\star t : E$ and $C^*_\Sigma \subseteq F$ and $\Sigma \vdash t \Downarrow^?_F a$, then evaluation does not get stuck and $\exists ans, \Sigma' \supseteq \Sigma$ such that $a = \langle \Sigma', ans \rangle$ and $F; \Sigma'; \emptyset \vDash_a ans : E$.*

The proof is by induction on the evaluation derivation, using inversion of the relaxed typing judgment in each case and the substitution and renaming lemmas of the mechanization to push typing through the value substitutions performed by the rules. The full development is mechanized (theorem `Eval.safety`, Appendix B).

Transporting the theorem along the round trip of Appendix A.2 yields safety for the MNF system of the main text.

Theorem A.6 (MNF Safety). *If* $C; \Sigma; \emptyset \vdash t : E$ *and* $C^*_\Sigma \subseteq F$ *and* $\Sigma \vdash t \Downarrow_F \langle \Sigma', ans \rangle$, *then* $\Sigma' \supseteq \Sigma$ *and*

1. *if* $ans = v$ *then* $\emptyset; \Sigma'; \emptyset \vdash \widehat{v} : E$,
2. *if* $ans = brk(\ell, v)$ *then* $\ell \in F$, $(\ell : S :: k) \in \Sigma'$, *and* $\emptyset; \Sigma'; \emptyset \vdash \widehat{v} : S$.

Starting from an MNF-typed term, evaluation produces an answer whose payload normalizes to an MNF-typed value of the expected type. The main text states the relaxed version as Theorem 4.1 in Section 4.

A.5 Corollaries

The safety theorem yields the runtime guarantees stated in Section 4. We restate them here in full.

Corollary A.7 (Effect Safety). *If* $\emptyset; \emptyset; \emptyset \vdash_\star t : T \wedge \emptyset$ *and* $\emptyset \vdash t \Downarrow_\emptyset \langle \Sigma', ans \rangle$, *then* $ans = v$ *for some value* v.

A well-typed closed program with empty capture always returns a value, no break goes unhandled.

Corollary A.8 (Used Label Prediction). *If* $C; \Sigma; \emptyset \vdash t : E$ *and* $\Sigma \vdash t \Downarrow^?_{C^*_\Sigma} a$, *then evaluation does not get stuck.*

That is, evaluation of t relies only on access to runtime labels of C. In other words, the typing judgment's use set correctly predicts the set of labels used during evaluation of t.

Corollary A.9 (Capture Prediction). *If* $D; \Sigma; \emptyset \vdash t : S \wedge C$ *and* $\Sigma \vdash t \Downarrow_{D^*_\Sigma} \langle \Sigma', v \rangle$, *then*

$$\Sigma'; \emptyset \vdash \lfloor v \rfloor <: C \qquad \text{and} \qquad \lfloor v \rfloor^*_{\Sigma'} \subseteq C^*_{\Sigma'}$$

Recall that a potential use set $\lfloor v \rfloor$ bounds the captures that a value reaches upon usage. We can predict the potential use set of the answer returned by the evaluation of a term from its static typing judgment. This is similar to the Capture Prediction for Terms lemma in $CC_{<:\square}$ [6].

Corollary A.10 (Closure Tightness). *If* $\emptyset; \Sigma; \emptyset \vdash_\star v_f : (\forall (z : T \wedge C_1) E) \wedge C_0$, $\emptyset; \Sigma; \emptyset \vdash_\star v_x : T \wedge C_2$, $\Sigma; \emptyset \vdash C_2 <: C_1$, *and* $\Sigma \vdash v_f\ v_x \Downarrow a$, *then* $\Sigma \vdash v_f\ v_x \Downarrow_{(C_0)^*_\Sigma \cup (C_2)^*_\Sigma} a$.

Applying a closure to a fitting argument needs access to exactly the labels predicted by the two capture sets and nothing more. A closure's annotation is thus a tight interface to its runtime behavior.

Corollary A.11 (Handler Coverage). *If* $D; \Sigma; \Gamma \vdash t : E$ *and* $\Sigma \vdash t \Downarrow_{D^*_\Sigma} \langle \Sigma', brk(\ell, v) \rangle$, *then:*

1. Breaks. $\ell \in D^*_{\Sigma'}$.
2. Intercepts. *If* $t = intercept[R, C_t, \phi]$ *with* h *in* t', *and* $(\ell : S :: k) \in \Sigma$, *then either* h *was evaluated, or* $k \in \top \setminus \phi$.

No label can be used at runtime that was not statically accounted for. An escaping break was visible in the use set, and an unhandled label passing through an intercept fits the required shape. Note that the second point is trivial from the evaluation rules.

Corollary A.12 (Boundary Safety). *If* $C; \Sigma; \emptyset \vdash_{\star}$ *boundary*$[S, k]$ *as* $\langle c, x\rangle$ *in* $t : E$ *and* $\Sigma \vdash$ *boundary*$[S, k]$ *as* $\langle c, x\rangle$ *in* $t \Downarrow_{C^*_\Sigma} \langle \Sigma', v\rangle$*, then the fresh label* ℓ *allocated by the boundary satisfies* $\ell \notin \lfloor v \rfloor^*_{\Sigma'}$.

A boundary label is local to its scope. It cannot occur in the potential use set of any value that leaves the boundary, even nested inside closures.

Corollary A.13 (Pass-Handler Safety). *Let* $t =$ *intercept*$[R, C_t, \phi]$ *with* h *in* t_0 *be typed by* (rt-icp-pass) *with use set* $C = C_t.proj[\top \setminus \phi] \cup C_h$*, and suppose the handler's captures exclude the intercepted kind,* $\Sigma; \emptyset \vdash C_h <: C_h.proj[\top \setminus \phi]$*. If* $\Sigma \vdash t \Downarrow_{C^*_\Sigma} \langle \Sigma', brk(\ell, v)\rangle$ *with* $(\ell : S :: k) \in \Sigma'$*, then* $k \notin \phi$.

When a pass handler does not itself retain labels of the intercepted kind, the intercept is exhaustive for that kind. No break classified under ϕ can escape it.

B The Lean 4 Mechanization

All definitions and results of Appendix A are mechanized in Lean 4. The development is called CaplessK, comprises about 10 000 lines of Lean in 59 files, and is free of `sorry` placeholders and of axioms beyond Lean's kernel. This section is a guide for readers who want to relate the pencil and paper formulation to the mechanized one.

B.1 Proof Structure

The central mechanized result is type safety for MNF terms, obtained by the round trip described in Appendix A.

(1) **MNF typing to relaxed typing.** The inductive `MNF.HasTyp` (`MNF/Typing.lean`) mirrors the typing figures of the main text. Lemma A.1 is `HasTyp.from_mnf_typing`: every `MNF.HasTyp` derivation is a `HasTyp` derivation of the relaxed system (`Typing.lean`).

(2) **Type-safe evaluation.** The fuel-indexed evaluator `Term.eval` (`Eval.lean`) implements checked big-step evaluation of relaxed terms. Theorem A.5 is `Eval.safety` (`Safety.lean`), proved by well-founded induction on fuel with the four invariants of the theorem statement maintained through every recursive call. Each case applies an inversion lemma (`Inversion/Typing.lean`) to decompose the typing derivation, the induction hypothesis to each sub-evaluation, and the structural lemmas `HasTyp.subst` and `HasTyp.rename` to reassemble the result.

(3) **Back to MNF.** The normalization function `Term.as_mnf` (`MNF/Core.lean`) implements Definition A.3. Lemma A.4 combines `Term.as_mnf.is_mnf` and `HasTyp.as_mnf`, and Lemma A.2 is `MNF.HasTyp.from_regular_typing`. Chaining all three steps gives Theorem A.6 as `MNF.safety` (`MNF/Basic.lean`).

The corollaries of Appendix A.5 are derived from `Eval.safety` in `Safety/Corollaries.lean`.

B.2 Substitutions

Every substitution performed by the evaluation rules is an instance of a single structure `Subst` (`Subst.lean`) that bundles four simultaneous maps.

`rebind` sends a term variable to a term.

`rebind_capt` sends a term variable to a capture set, its capture annotation.

`trebind` sends a type variable to a shape type.

`crebind` sends a capture variable to a capture set.

The two maps on term variables are separate because a term variable x occurs in two syntactic roles, as a term and as the capture annotation $\{x\}$ inside types and capture sets. Both roles must be rewritten together to preserve typing, which is why the evaluation rules of Appendix A.3 always pair a term substitution $[v/x]$ with a capture substitution $[\lfloor v \rfloor/x]$. The named substitutions used by the rules are the following instances.

`Subst.open` v C Term variable 0 becomes v as a term and C as an annotation, `rebind`$(0) = v$ and `rebind_capt`$(0) = C$. This is the substitution of (E-APP), (E-LET-V) and the invoke rules, with C the effective use set of v.

`Subst.exlet_open` D v C Additionally sends capture variable 0 to the witnessed set, `crebind`$(0) = D$, and is used by (E-EX-V).

`Subst.boundary_open` At boundary entry a fresh label ℓ replaces the break-return variable x both as a term and as its annotation, and the classifier variable c, so `rebind`$(0) = \ell$, `rebind_capt`$(0) = \{\ell\}$ and `crebind`$(0) = \{\ell\}$. This is the $[\ell/c][\ell/x][\ell/x]$ of the boundary rules.

`Subst.handler_open` ℓ ϕ S v At an intercept match the four handler binders are filled in, the break value $y \mapsto v$, the break label $b \mapsto \ell$, the shape variable $X \mapsto S$ and the capture set $c \mapsto \{(\ell \mid \phi)\}$, matching the substitution in (E-ICP-M).

Narrowing is not a separate lemma. The three instances `CtxSubst.narrow`, `.tnarrow` and `.cnarrow` express it as an identity substitution from a context with a weaker bound into the same context with a stronger bound, so replacing a bound by a subtype follows from the substitution lemma below.

B.3 Structural and Inversion Lemmas

The structural lemmas are stated through two predicates that abstract the notion of a well-typed context morphism. CtxRename(Γ, f, Δ) (`Renaming/Morphism.lean`) holds when the renaming f carries every term, type, capture and label binding of Γ to the corresponding binding of Δ. CtxSubst(Γ, s, Δ) (`Subst/Morphism.lean`) holds when the substitution s is admissible from Γ to Δ, meaning every substituted term is well-typed at the substituted binding type, capture annotations remain consistent under subcapturing, and type and capture variable images satisfy the respective subtyping and bound constraints. Admissibility carries one extra requirement, `var_or_val`, that each substituted term is a variable or a value. The safety proof relies on it because it substitutes evaluation results, which are values, without a term store.

Lemma B.1 (Renaming, `HasTyp.rename`). *If* $C; \Sigma; \Gamma \vdash_\star t : E$ *and* CtxRename(Γ, f, Δ)*, then* $C[f]; \Sigma; \Delta \vdash_\star t[f] : E[f]$. *The analogue holds for subtyping, subcapturing and kinding.*

Lemma B.2 (Substitution, `HasTyp.subst`). *If* $C; \Sigma; \Gamma \vdash_\star t : E$ *and* CtxSubst(Γ, s, Δ)*, then* $C[s]; \Sigma; \Delta \vdash_\star t[s] : E[s]$. *The analogue holds for subtyping, subcapturing and kinding.*

Inversion lemmas (`Inversion/Typing.lean`, `Inversion/Subtyping.lean`) decompose a typing or subtyping derivation for a fixed syntactic form into its premises up to subsumption. Table 1 lists them. Two further lemmas single out values. `HasTyp.value_use_set_is_empty` states that a well-typed value can always be retyped with an empty use set, since a value uses no labels at the time it is formed. `HasTyp.tighten_value` (`Eval/Props.lean`) states that a value typed at $S \wedge C$ can be retyped at $S \wedge \lfloor v \rfloor$ with $\Sigma; \Gamma \vdash \lfloor v \rfloor <: C$, so its potential use set $\lfloor v \rfloor$ is the tightest capture annotation it admits. These are the lemmas behind Closure Tightness (Corollary A.10) and Boundary Safety (Corollary A.12).

Table 1. Inversion lemmas of the mechanization.

Lemma	**Form inverted**
`HasTyp.var_inv`	$C; \Sigma; \Gamma \vdash_\star x : E$
`HasTyp.abs_inv`	$C; \Sigma; \Gamma \vdash_\star \lambda^{C_0}(x{:}T)\ t : E$
`HasTyp.tabs_inv`	$C; \Sigma; \Gamma \vdash_\star \lambda^{C_0}[X{<:}S]\ t : E$
`HasTyp.cabs_inv`	$C; \Sigma; \Gamma \vdash_\star \lambda^{C_0}[c{:}B]\ t : E$
`HasTyp.app_inv`	$C; \Sigma; \Gamma \vdash_\star t\ s : E$ (application or invoke)
`HasTyp.tapp_inv`	$C; \Sigma; \Gamma \vdash_\star t[S] : E$
`HasTyp.capp_inv`	$C; \Sigma; \Gamma \vdash_\star t[D] : E$
`HasTyp.lett_inv`	$C; \Sigma; \Gamma \vdash_\star \mathsf{let}\ x = t\ \mathsf{in}\ u : E$
`HasTyp.exlet_inv`	$C; \Sigma; \Gamma \vdash_\star \mathsf{let}\ \langle c, x \rangle = t\ \mathsf{in}\ u : E$
`HasTyp.pack_inv`	$C; \Sigma; \Gamma \vdash_\star \mathsf{pack}\ \langle D, t \rangle : E$
`HasTyp.boundary_inv`	$C; \Sigma; \Gamma \vdash_\star \mathsf{boundary}[S, k] \cdots : E$
`HasTyp.intercept_inv`	$C; \Sigma; \Gamma \vdash_\star \mathsf{intercept}[\dots]\ \mathsf{with}\ h\ \mathsf{in}\ t : E$
`HasTyp.label_inv`	$C; \Sigma; \Gamma \vdash_\star \ell : E$
`SSubtyp.top_inv`	$\Sigma; \Gamma \vdash \top <:_s S$
`SSubtyp.break_inv`	$\Sigma; \Gamma \vdash \mathsf{Break}\ [S_1] <:_s S$
`SSubtyp.ffun_inv`	$\Sigma; \Gamma \vdash (\forall(x{:}T_1)E_1) <:_s S$
`SSubtyp.tfun_inv`	$\Sigma; \Gamma \vdash (\forall[X{<:}S_1]E_1) <:_s S$
`SSubtyp.cfun_inv`	$\Sigma; \Gamma \vdash (\forall[c{:}B_1]E_1) <:_s S$

B.4 Semantic Predicates

Three predicates carry the invariants of the safety proof. The mechanization separates the runtime store φ, which records the shape type and classifier of each allocated label, from the static context Γ. In the notation of Appendix A the store φ and the label bindings of Γ are the single label context Σ.

SemStore(Γ, φ) (`Store/Props.lean`) The store is consistent with the context, every label ℓ satisfies $(\ell : \varphi.\mathsf{type}(\ell) :: \varphi.\mathsf{class}(\ell)) \in \Gamma$.

TypedAnswer$(\Gamma, \varphi, F, a, E)$ (`Sem.lean`) The answer is well-typed. A value answer v satisfies $\emptyset; \Gamma \vdash_\star v : E$. A break answer $\mathsf{brk}(\ell, v)$ satisfies $\ell \in F$ and $\emptyset; \Gamma \vdash_\star v : \varphi.\mathsf{type}(\ell)$, typing the payload at the label's stored shape type with empty capture. This is the answer typing $F; \Sigma; \Gamma \vDash_{\mathsf{a}} a : E$ of Figure 9.

TypedRes$(\Gamma, \varphi, F, \mathit{res}, E)$ (`Sem.lean`) The evaluation result is well-typed up to a label renaming f, there are Γ' and φ' with $\mathit{res} = \mathsf{ok}(f, a, \varphi')$, CtxRename$(\Gamma, f, \Gamma')$, SemStore$(\Gamma', \varphi')$, and TypedAnswer$(\Gamma', \varphi', f(F), a, E[f])$. In particular $\mathit{res} \neq \mathsf{stuck}$.

The renaming f accommodates the fresh labels allocated by boundaries, which grow the label index space, and is the mechanized counterpart of the paper convention that the label context only grows.

B.5 Type Safety

In Theorem A.5 the single label context Σ serves at once as the static context of the typing judgment and the runtime store of the evaluator, so their agreement holds by construction. The mechanization separates the static context Γ from the runtime store φ, so `Eval.safety` (`Safety.lean`) carries the extra store-consistency premise SemStore(Γ, φ), premise (4) below, that Theorem A.5 leaves implicit. Its conclusion TypedRes (Appendix B.4) likewise replaces the growing-context convention $\Sigma' \supseteq \Sigma$ with an explicit label renaming and records that evaluation does not get stuck. Otherwise the statement is Theorem A.5.

$$\underbrace{\varphi \vdash t \Downarrow_F res}_{\text{(1) evaluation terminates}} \wedge \underbrace{C; \Gamma \vdash_\star t : E}_{\text{(2) } t \text{ is well typed}} \wedge \underbrace{C^*_\varphi \subseteq F}_{\text{(3) the allowance covers the use set}} \wedge \underbrace{\mathsf{SemStore}(\Gamma, \varphi)}_{\text{(4) store consistent with } \Gamma} \implies \mathsf{TypedRes}(\Gamma, \varphi, F, res, E)$$

Premise (1) is expressed as $\mathtt{eval}(t, \varphi, F, fuel) = \mathtt{some}(res)$ for some finite fuel, and the proof is by well-founded induction on that fuel, maintaining all four premises across every recursive call. Each case runs in three steps.

(1) **Inversion.** The inversion lemma for the term's form (Table 1) decomposes premise (2) into sub-term typings, and premise (3) is propagated to the sub-terms using subcapturing and the structure of the evaluation rule.
(2) **Induction.** The induction hypothesis applies to each sub-evaluation at strictly smaller fuel, yielding a TypedRes for each sub-result.
(3) **Reassembly.** The sub-results recombine into a TypedRes for the whole term. `HasTyp.subst` (Lemma B.2) discharges the value substitutions of (E-APP), (E-LET-V), (E-EX-V) and the boundary and intercept rules, and `HasTyp.rename` (Lemma B.1) transports typing across the label growth at (E-BND-V), (E-BND-C) and (E-BND-P).

Chaining the round trip of Appendix A.2 around this theorem gives `MNF.safety`, the mechanized form of Theorem A.6.

B.6 Module Overview

Table 2. Modules of the CaplessK development.

Module	**Contents**
`Classifier/*`	classifier tree, kinds, intersection, subtraction, subkinding, disjointness, inclusion, set semantics
`CaptureSet`, `CaptureSet/*`	capture sets, projection, renaming, substitution
`Term`, `Term/*`	de Bruijn terms, values, potential use sets
`Type`, `Type/*`	shape, capturing and existential types
`Context`	typing contexts with four binding forms
`Subcapt`	subcapturing and capture kinding
`Subtyping`, `Subtyping/*`	subtyping and its transitivity
`Typing`	the relaxed type system (HasTyp)
`MNF/*`	MNF predicate, normalization, MNF typing, MNF safety
`Renaming/*`, `Subst/*`	context morphisms and the structural lemmas
`Inversion/*`	inversion lemmas for typing and subtyping
`Store`, `Store/*`	label stores and their consistency with contexts
`Filter`, `Filter/*`	runtime label allowances
`Eval`, `Eval/*`	the checked big-step evaluator and its properties
`Sem`	answer and result typing
`Safety`, `Safety/*`	the safety theorem and its corollaries
`Try`	the mechanized Try encoding of Section 3
`FinFun`, `Set`, `Tactics`	infrastructure

Table 2 maps the modules of the development to their contents. All paths are relative to the `CaplessK/` source directory.

B.7 Differences from the Paper Presentation

The mechanization makes nine representation choices that differ from the pencil and paper formulation. None of them affects the meaning of the theorems.

- **de Bruijn indices.** Terms are intrinsically scoped over a quadruple index counting term variables, type variables, capture variables and labels. Wellformedness premises of the paper rules are subsumed by this indexing and do not appear in the mechanization.
- **Labels live in the context.** The paper separates the label context Σ from Γ. The mechanization uses a single context with label bindings, and a predicate `SemStore` states that these bindings agree with the runtime label store. In the paper formulation this agreement holds by construction because the same Σ appears in typing and evaluation.
- **Explicit label renamings.** Where the paper lets the label context grow monotonically, the evaluator returns an injective renaming between the initial and final label index spaces, and the safety proof transports typing along it (`CtxRename`, `HasTyp.rename`).
- **Functional, fuel-indexed evaluation.** `Term.eval` is a total function on a fuel argument returning either an answer or an explicit `stuck` marker. The relational judgment $\Sigma \vdash t \Downarrow_F a$ of the paper corresponds to `Term.eval` returning an answer at some fuel, and Corollary A.9 appears as the statement that evaluation of a well-typed term never returns `stuck`.
- **Annotated applications and packing.** Application nodes in the mechanization carry the use set of the argument, and intercept nodes carry the static capture sets of handler and body. The paper instead reads the potential use set $\lfloor v \rfloor$ off the evaluated operand. The typing rules constrain the stored annotations to coincide with the sets the paper computes, so the two formulations check the same labels.
  Similarly, packing stores the capture set of the value during typing as part of its term, which gets substituted during unpacking instead of $\lfloor v \rfloor$.
- **Handlers as open terms.** The mechanization types the handler body in a context extended with its four binders, whereas the paper packages the handler as a value of the Handler type abbreviation. The two presentations are interderivable by abstracting and applying the handler.
- **Substitutions in bulk.** A single structure `Subst` bundles the term, capture annotation, type and capture variable substitutions, with admissibility captured by the `CtxSubst` predicate. The simultaneous substitutions of the evaluation rules, such as $[\lfloor v \rfloor / x][v/x]$, are single instances of this structure, and one lemma `HasTyp.subst` covers all of them. Narrowing is the special case of an identity substitution into a context with a stronger bound.
- **Split subtyping.** The single subtyping judgment of Figure A.1 is split into three mutually defined judgments for shape, capturing and existential types (`SSubtyp`, `CSubtyp`, `ESubtyp`).
- **Algorithmic kinding.** The mechanized capture kinding has no counterpart of (k-sub), subkind reasoning is inlined into the rule for kind-bounded capture variables, and the general rule is derived as a lemma (`CaptureKind.sub`). Similarly, (sc-proj) is mechanized for singleton sets only, with the general form derived by induction.

B.8 Correspondence of Rules and Theorems

Table 3 relates the judgments of the paper to the corresponding Lean types, and Table 4 locates each stated result in the sources. The constructors of `HasTyp` correspond one to one to the rules of Figure A.3: `var`, `sub`, `abs`, `tabs`, `cabs`, `app`, `tapp`, `capp`, `lett`, `exlet`, `pack_var`, `pack_val`, `label`,

Table 3. Correspondence between paper judgments and Lean types.

Paper	**Lean type**	**Source file**
$k_1 \sqsubseteq k_2$	`Classifier.Subclass`	`Classifier/Core.lean`
$\phi_1 \wedge \phi_2$, $\phi_1 \setminus \phi_2$	`Kind.Intersect`, `Kind.Subtract`	`Classifier/*.lean`
ϕ **empty**, $\phi_1 \preceq \phi_2$, $k \in \phi$	`Kind.IsEmpty`, `Kind.Subkind`, `Kind.Contains`	`Classifier/*.lean`
$C_1 \sqsubseteq C_2$	`CaptureSet.Subset`	`CaptureSet.lean`
$\Sigma; \Gamma \vdash C : \phi$	`CaptureKind`	`Subcapt.lean`
$\Sigma; \Gamma \vdash C_1 <: C_2$	`Subcapt`	`Subcapt.lean`
$\Gamma \vdash B_1 <:_{\mathrm{B}} B_2$	`Subbound`	`Subtyping.lean`
$\Sigma; \Gamma \vdash E_1 <: E_2$	`SSubtyp`, `CSubtyp`, `ESubtyp`	`Subtyping.lean`
$C; \Sigma; \Gamma \vdash t : E$ (MNF)	`MNF.HasTyp`	`MNF/Typing.lean`
$C; \Sigma; \Gamma \vdash_\star t : E$	`HasTyp`	`Typing.lean`
t in MNF, $\widehat{t}$	`Term.IsMNF`, `Term.as_mnf`	`MNF/Core.lean`
$\Sigma \vdash t \Downarrow_F a$	`Term.eval`	`Eval.lean`
C^*_Σ, $\lfloor v \rfloor$	`CaptureSet.runtime_labels`, `Value.use_set`	`Eval.lean`, `Term/Ops.lean`
$F; \Sigma; \Gamma \vDash_{\mathrm{a}} a : E$	`TypedAnswer`	`Sem.lean`

Table 4. Correspondence between paper results and Lean theorems.

Result	**Lean theorem**	**Source file**
Lemma A.1	`HasTyp.from_mnf_typing`	`MNF/Typing.lean`
Lemma A.2	`MNF.HasTyp.from_regular_typing`	`MNF/Typing.lean`
Lemma A.4	`Term.as_mnf.is_mnf`, `HasTyp.as_mnf`	`MNF/Core.lean`
Theorem A.5	`Eval.safety`	`Safety.lean`
Theorem A.6	`MNF.safety`	`MNF/Basic.lean`
Corollary A.7	`Eval.effect_safety`	`Safety/Corollaries.lean`
Corollary A.9	`Eval.capture_prediction`	`Safety/Corollaries.lean`
Corollary A.10	`Eval.closure_tightness`	`Safety/Corollaries.lean`
Corollary A.11	`Eval.handler_coverage`	`Safety/Corollaries.lean`
Corollary A.12	`Eval.boundary_safety`	`Safety/Corollaries.lean`
Corollary A.13	`Eval.pass_handler_safety`	`Safety/Corollaries.lean`
Set semantics of kinds (Section 4.1)	equivalence lemmas	`Classifier/Semantics.lean`

`invoke`, `boundary`, `intercept_pass` and `intercept_gen` implement (RT-VAR), (RT-SUB), (RT-ABS), (RT-TABS), (RT-CABS), (A-RT-APP), (RT-TAPP), (RT-CAPP), (RT-LET), (RT-EX), (A-RT-PACK-VAR), (A-RT-PACK-VAL), (RT-LABEL), (RT-BREAK), (RT-BND), (RT-ICP-PASS) and (RT-ICP-GEN) respectively. The cases of `Term.eval` implement the evaluation rules of Figures A.4 and 8 in the same way, and the constructors of the kind algebra carry the names of the rules in Figure A.2.

B.9 The Try Example

The encoding of `Try[T]` from Section 3 is mechanized in full in `Try.lean`, from the Church encoded type down to the constructor and its typing derivation. We give the development here in the notation of the paper.

Comparison to Scala's `Try`. Scala's `Try[T]` constructor has the capture-typed signature

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

so the returned `Try[T]` captures only the control-kinded effects of `body`. The encoding matches this exactly. `Try.mk` produces a value of type $\mathsf{Try}(X, \{(c \mid \mathsf{Control})\})$, where only the control-kinded part of the thunk's capture set $\{c\}$ is retained. Although a Try value is itself pure, its own use set is empty, applying it, that is pattern matching on the result, charges the retained control captures to the caller through the capture annotation on the outer binder of its type. The control effects therefore resurface at the use site, which is the intended behavior of a value that stands for a suspended control effect.

The Try type. Given a shape type T and a capture set C, the type $\mathsf{Try}(T, C)$ is the Church encoded sum of a success and a failure case (`Try`).

$$\begin{aligned}\mathsf{Try}(T, C) \;=\; & \forall[X{<:}\top].\ \forall[c{:}\top].\\ & \Big(\forall\big(s{:}\ \underbrace{(\forall(x{:}T)\,X)\,\hat{}\,\{c\}}_{\text{success continuation}}\big).\\ & \quad \forall\big(f{:}\ \underbrace{(\forall[Y{<:}\top].\ \forall(b{:}(\mathsf{Break}\ [Y])\,\hat{}\,C)\ (\forall(r{:}Y)\,X)\,\hat{}\,\{c\})\,\hat{}\,\{c\}}_{\text{failure continuation}}\big).\ X\Big)\,\hat{}\,\{c \mid \top\} \cup C\end{aligned}$$

A value of type $\mathsf{Try}(T, C)$ is a function that, given a success continuation and a failure continuation, invokes the appropriate one. The failure continuation receives the break payload b together with a resumption r. The type is proved covariant in both T and C (`Try.sub`), so a Try over a smaller result type and a smaller capture set is a subtype.

Constructors. `Try.success` takes a value $v : T$ and produces an MNF term of type $\mathsf{Try}(T, C)$ that ignores the failure continuation and applies the success continuation to v (`Try.success.typeable`). `Try.failure` takes a label $\ell : (\mathsf{Break}\ [U])\,\hat{}\,C$ and a value $v : U$ and produces an MNF term of type $\mathsf{Try}(T, C)$ that ignores the success continuation and passes ℓ and v to the failure continuation (`Try.failure.typeable`).

The constructor `Try.mk`. The main entry point has type

$$\mathsf{Try.mk} : \forall[X{<:}\top].\ \forall[c{:}\top]^{\{(c|\top\setminus\mathrm{Control})\}}.\ \forall\big(t{:}(\forall(x{:}\top)\,X)\,\hat{}\,\{c\}\big).\ \mathsf{Try}(X, \{(c \mid \mathsf{Control})\}).$$

It takes a thunk t and proceeds in three steps (`Try.mk`, `Try.mk'`).

(1) It runs the thunk inside an intercept that intercepts control-kinded breaks, that is $\phi = \mathsf{Control}$.
(2) If the thunk returns a value w, it wraps w with `Try.success` into a success result.
(3) If a break $\mathsf{brk}(\ell, w)$ escapes the thunk, the intercept handler invokes `Try.failure` with ℓ and w.

The capture set of the resulting Try value is $C.\mathsf{proj}[\mathsf{Control}]$, the control-kinded part of the thunk's capture set C, reflecting that only control-kinded captures can surface as failures. Because the handler is a pass handler, the use set of the whole term is the non-control part $C.\mathsf{proj}[\top \setminus \mathsf{Control}]$ (`Try.mk'.typeable`). The entire term is in MNF and is proved well-typed in the MNF system (`Try.mk.typeable`), so it is a genuine program of System Capless($\mathcal{K}$) and not only of the relaxed system.

B.10 Checking the Development

The development builds with Lean 4.30.0 using `lake build`. The main safety theorems are in `Safety.lean` and `MNF/Basic.lean`, and the corollaries in `Safety/Corollaries.lean`. The sources contain no `sorry` and declare no axioms, so the kernel checks every result down to Lean's foundations.