Keeping Track of Capabilities

MARTIN ODERSKY, EPFL
ALEKSANDER BORUCH-GRUSZECKI, EPFL
EDWARD LEE, University of Waterloo
JONATHAN BRACHTHÄUSER, Eberhard Karls University of Tübingen
ONDŘEJ LHOTÁK, University of Waterloo

Type systems usually characterize the shape of values but not their free variables. However, many desirable safety properties could be guaranteed if one knew the free variables captured by values. We describe $CC_{<:\square}$, a calculus where such captured variables are succinctly represented in types, and show it can be used to safely implement effects and effect polymorphism via *scoped capabilities*. We discuss how the decision to track captured variables guides key aspects of the calculus, and show that $CC_{<:\square}$ admits simple and intuitive types for common data structures and their typical usage patterns. We demonstrate how these ideas can be used to guide the implementation of capture checking in a practical programming language.

1 INTRODUCTION

Effects are aspects of computation that go beyond describing shapes of values and that we still want to track in types. What exactly is modeled as an effect is a question of language or library design. Some possibilities are: *reading* or *writing* to mutable state outside a function, *throwing an exception* to signal abnormal termination of a function, *I/O* including file operations, network access, or user interaction, *non-terminating* computations, *suspending* a computation e.g., waiting for an event, or *using a continuation* for control operations.

Despite hundreds of published papers there is comparatively little adoption of static effect checking in programming languages. The few designs that *are* widely implemented (for instance Java's checked exceptions or monadic effects in some functional languages) are often critiqued for being both too verbose and too rigid. The problem is not lack of expressiveness – systems have been proposed and implemented for many quite exotic kinds of effects. Rather, the problem is simple lack of usability and flexibility, with particular difficulties in describing polymorphism. This leads either to overly complex definitions, or to the requirement to duplicate large bodies of code.

Classical type-systematic approaches fail since effects are inherently transitive along the edges of the dynamic call-graph: A function's effects include the effects of all the functions it calls, transitively. Traditional type and effect systems have no lightweight mechanism to describe this behavior. The standard approach is either manual specialization along specific effect classes, which means large-scale code duplication, or quantifiers on all definitions along possible call graph edges to account for the possibility that some call target has an effect, which means large amounts of boilerplate code. Arguably, it is this problem more than any other that has so far hindered wide scale application of effect systems.

A promising alternative that circumvents this problem is to model effects via capabilities [Brachthäuser et al. 2020a; Gordon 2020; Liu 2016; Marino and Millstein 2009; Osvald et al. 2016]. Capabilities exist in many forms, but we will restrict the meaning here to simple object capabilities

Authors' addresses: Martin Odersky, EPFL, martin.odersky@epfl.ch; Aleksander Boruch-Gruszecki, EPFL, aleksander.boruch-gruszecki@epfl.ch; Edward Lee, University of Waterloo, e45lee@uwaterloo.ca; Jonathan Brachthäuser, Eberhard Karls University of Tübingen, jonathan.brachthaeuser@uni-tuebingen.de; Ondřej Lhoták, University of Waterloo, olhotak@uwaterloo.ca.

2023. 0164-0925/2023/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

represented as regular program variables. For instance, consider the following two formulations of a method in Scala[1] which are morally equivalent:

```
def f(): T throws E
def f()(using ct: CanThrow[E]): T
```

The first version looks like it describes an effect: Function `f` returns a `T`, or it might throw exception `E`. The effect is mentioned in the return type `throws[T, E]` where the `throws` is written infix.

The second version expresses analogous information as a capability: Function `f` returns a value of type `T`, *provided* it can be passed a capability `ct` of type `CanThrow[T]`. The capability is modelled as a parameter. To avoid boilerplate, that parameter is synthesized automatically by the compiler at the call site assuming a matching capability is defined there. This is expressed by a `using` keyword, which indicates that a parameter is implicit in Scala 3 (Scala 2 would have used the `implicit` keyword instead). The fact that capabilities are implicit rather than explicit parameters helps with conciseness and readability of programs, but is not essential for understanding the concepts discussed in this paper.

Aside: The link between the "effect" and the "capability" version of `f` can be made more precise by means of implicit function types [Odersky et al. 2018]. It is embodied in the following definition of the `throws` type:

```
infix type throws[T, E <: Exception] = CanThrow[E] ?=> T
```

The implicit function type `CanThrow[E] ?=> T` represents functions from `CanThrow[E]` to `T` that are applied implicitly to arguments synthesized by the compiler. This gives a direct connection between the effect view based on the `throws` type and the capability view based on its expansion.

An important benefit of this switch from effects to capabilities is that it gives us polymorphism for free. For instance, consider the `map` function in class `List[A]`. If we wanted to take effects into account, it would look like this:

```
def map[B, E](f: A -> B eff E): List[B] eff E
```

Here, `A -> B eff E` is hypothetical syntax for the type of functions from `A` to `B` that can have effect `E`. While looking reasonable in the small, this scheme quickly becomes unmanageable, if we consider that every higher-order function has to be expanded that way and, furthermore, that in an object-oriented language almost every method is a higher-order function [Cook 2009]. Indeed, many designers of programming languages with support for effect systems agree that programmers should ideally not be confronted with explicit effect quantifiers [Brachthäuser et al. 2020a; Leijen 2017; Lindley et al. 2017].

On the other hand, here is the type of `map` if we represent effects with capabilities.

```
def map(f: A => B): List[B]
```

Interestingly, this is exactly the same as the type of `map` in current Scala, which does not track effects! In fact, compared to effect systems, we now decompose the space of possible effects differently: `map` is classified as pure since it does not produce any effects in its own code, but when analyzing an application of `map` to some argument, the capabilities required by the argument are also capabilities required by the whole expression. In that sense, we get effect polymorphism for free.

The reason this works is that in an effects-as-capabilities discipline, the type `A => B` represents the type of *impure* function values that can *close over* arbitrary effect capabilities. (Alongside, we also define a type of pure functions `A -> B` that are not allowed to close over capabilities.)

[1]Scala 3.1 with language import `saferExceptions` enabled.

This seems almost too good to be true, and indeed there is a catch: It now becomes necessary to reason about capabilities captured in closures, ideally by representing such knowledge in a type system.

```
class TooLarge extends Exception

def f(x: Int): Int throws TooLarge =
  if x < limit then x * x
  else throw TooLarge()

val xs: List[Int]
try xs.map(x => f(x))
catch case TooLarge => Nil
```

```
def f(x: Int)
      (using CanThrow[TooLarge]): Int =
  if x < limit then x * x
  else throw TooLarge()

val xs: List[Int]
try xs.map(x =>
      f(x)(using new CanThrow[TooLarge]))
catch case TooLarge => Nil
```

Fig. 1. Exception handling: source (left) and compiler-generated code (right)

To see why, consider that effect capabilities are often scoped and therefore have a limited lifetime. For instance a `CanThrow[E]` capability would be generated by a `try` expression that catches `E`. It is valid only as long as the `try` is executing. Figure 1 shows an example of capabilities for checked exceptions, both as source syntax on the left and with compiler-generated implicit capability arguments on the right. The following slight variation of this program would throw an unhandled exception since the function `f` is now evaluated only when the iterator's `next` method is called, which is after the **try** handling the exception has exited.

```
val it =
  try xs.iterator.map(f)
  catch case TooLarge => Iterator.empty
it.next()
```

A question answered in this paper is how to rule out `Iterator`'s lazy `map` statically while still allowing `List`'s strict `map`. A large body of research exists that could address this problem by restricting reference access patterns. Relevant techniques include linear types [Wadler 1990], rank 2 quantification [Launchbury and Sabry 1997], regions [Grossman et al. 2002; Tofte and Talpin 1997], uniqueness types [Barendsen and Smetsers 1996], ownership types [Clarke et al. 1998; Noble et al. 1998], and second class values [Osvald et al. 2016]. A possible issue with many of these approaches is their relatively high notational overhead, in particular when dealing with polymorphism.

The approach we pursue here is different. Instead of restricting certain access patterns a priori, we focus on describing what capabilities are possibly captured by values of a type. At its core there are the following two interlinked concepts:

- A *capturing type* is of the form $\{c_1, \ldots, c_n\}\ T$ where T is a type and $\{c_1, \ldots, c_n\}$ is a *capture set* of capabilities.
- A *capability* is a parameter or local variable that has as type a capturing type with non-empty capture set. We call such capabilities *tracked* variables.

Every capability gets its authority from some other, more sweeping capabilities which it captures. The most sweeping capability, from which ultimately all others are derived, is "$*$", the *universal capability*.

As an example how capabilities are defined and used, consider a typical *try-with-resources* pattern:

```
def usingFile[T](name: String, op: ({*} OutputStream) => T): T =
  val f = new FileOutputStream(name)
  val result = op(f)
```

```
    f.close()
    result

val xs: List[Int] = ...
def good = usingFile("out", f => xs.foreach(x => f.write(x)))
def fail =
  val later = usingFile("out",
        f => (y: Int) => xs.foreach(x => f.write(x + y)))
  later(1)
```

The `usingFile` method runs a given operation `op` on a freshly created file, closes the file, and returns the operation's result. The method enables an effect (writing to a file) and limits its validity (to until the file is closed). Function `good` invokes `usingFile` with an operation that writes each element of a given list `xs` to the file. By contrast, function `fail` represents an illegal usage: It invokes `usingFile` with an operation that returns a function that, when invoked, will write list elements to the file. The problem is that the writing happens in the application `later(1)` when the file has already been closed.

We can accept the first usage and reject the second by marking the output stream passed to `op` as a capability. This is done by prefixing its type with `{*}`. Our prototype implementation of capture checking will then reject the second usage with an error message.

This example used a capability parameter that was directly derived from `*`. But capabilities can also be derived from other non-universal capabilities. For instance:

```
def usingLogFile[T](f: {*} OutputStream, op: ({f} Logger) => T): T =
  op(Logger(f))
```

The `usingLogFile` method takes an output stream (which is a capability) and an operation, which gets passed a `Logger`. The `Logger` capability is derived from the output stream capability, as can be seen from its type `{f} Logger`.

This paper develops a *capture calculus*, $\mathsf{CC}_{<:\Box}$, as a foundational type system that allows reasoning about scoped capabilities. By sketching a prototype language design based on this calculus, we argue that it is expressive enough to support a wide range of usage patterns with very low notational overhead. The paper makes the following specific contributions.

- We define a simple yet expressive type system for tracking captured capabilities in types. The calculus extends $\mathsf{F}_{<:}$ with capture sets of capabilities.
- We prove progress and preservation of types relative to a small-step evaluation semantics. We also prove a capture prediction lemma that states that capture sets in types over-approximate captured variables in runtime values.
- We illustrate the practical applicability of the calculus with a number of examples that have been checked by a prototype capture checker implemented within the Scala 3 compiler.

The presented design is at the same time simple in theory and concise and flexible in its practical application. We demonstrate that the following elements are essential for achieving good usability:

- Use reference-dependent typing, where a formal function parameter stands for the potential references captured by its argument [Brachthäuser et al. 2022; Odersky et al. 2021]. This avoids the need to introduce separate binders for capabilities or effects. Technically, this means that references (but not general terms) can form part of types as members of capture sets. A similar approach is taken in the path-dependent typing discipline of DOT [Amin et al. 2016; Rompf and Amin 2016] and by reachability types for alias checking [Bao et al. 2021].

- Employ a subtyping discipline that mirrors subsetting of capabilities and that allows capabilities to be refined or abstracted. Subtyping of capturing types relies on a new notion of *subcapturing* that encompasses both subsetting (smaller capability sets are more specific than larger ones) and derivation (a capability singleton set is more specific than the capture set of the capability's type). Both dimensions are essential for a flexible modelling of capability domains.
- Limit propagation of capabilities in instances of generic types where they cannot be accessed directly. This is achieved by boxing types when they enter a generic context and unboxing on every use site [Brachthäuser et al. 2022].

Whereas many of our motivating examples describe applications in effect checking, the formal treatment presented here does not mention effects. In fact, the effect domains are intentionally kept open since they are orthogonal to the aim of the paper. Effects could be exceptions, file operations or region allocations, but also algebraic effects, IO, or any sort of monadic effects. To express more advanced control effects, one usually needs to add continuations to the operational semantics, or use an implicit translation to the continuation monad. In short, capabilities can delimit what effects can be performed at any point in the program but they by themselves don't perform an effect [Brachthäuser et al. 2020a; Gordon 2020; Liu 2016; Marino and Millstein 2009; Osvald et al. 2016]. For that, one needs a library or a runtime system that would be added as an extension of $CC_{<:\Box}$. Since $CC_{<:\Box}$ is intended to work with all such effect extensions, we refrain from adding a specific extension to its operational semantics.

The version of $CC_{<:\Box}$ presented here evolved from a system that was originally proposed to make exception checking safe [Odersky et al. 2021]. The earlier paper described a way to encode information about potentially raised exceptions as object capabilities passed in parameters. It noted that the proposed system is not completely safe since capabilities can escape in closures and it hypothesized a possible way to fix the problem by presenting a draft of what became $CC_{<:\Box}$. At the time, the meta theory of the proposed system was not worked out yet and the progress and preservation properties were left as conjectures. The present paper presents a fully worked-out meta theory with proofs of type soundness as well as a semantic characterization of capabilities. There are some minor differences in the operational semantics, which were necessary to make a progress theorem go through. We also present a range of use cases outside of exception handling, demonstrating the broad applicability of the calculus.

The rest of this paper is organized as follows. Section 2 explains and motivates the core elements of our calculus. Section 3 presents $CC_{<:\Box}$. Section 4 lays out its meta-theory. Section 5 illustrates the expressiveness of typing disciplines based on the calculus in examples. Section 6 discusses related work and Section 7 concludes.

2 INFORMAL DISCUSSION

This section motivates and discusses some of the key aspects of capture checking. All examples are written in an experimental language extension of Scala 3 [Scala 2022a] and were compiled with our prototype implementation of a capture checker [Scala 2022b].

2.1 Capability Hierarchy

We have seen in the intro that every capability except $*$ is created from some other capabilities which it retains in the capture set of its type. Here is an example that demonstrates this principle:

```
class FileSystem

class Logger(fs: {*} FileSystem):
```

```
    def log(s: String): Unit = ... // Write to a log file, using `fs`

  def test(fs: {*} FileSystem): {fs} LazyList[Int] =
    val l: {fs} Logger = new Logger(fs)
    l.log("hello world!")
    val xs: {l} LazyList[Int] =
      LazyList.from(1)
        .map { i =>
          l.log(s"computing elem # $i")
          i * i
        }
    xs
```

Here, the `test` method takes a `FileSystem` as a parameter. `fs` is a capability since its type has a non-empty capture set. The capability is passed to the `Logger` constructor and retained as a field in class `Logger`. Hence, the local variable `l` has type `{fs} Logger`: it is a `Logger` which retains the `fs` capability.

The second variable defined in `test` is `xs`, a lazy list that is obtained from `LazyList.from(1)` by logging and mapping consecutive numbers. Since the list is lazy, it needs to retain the reference to the logger `l` for its computations. Hence, the type of the list is `{l} LazyList[Int]`. On the other hand, since `xs` only logs but does not do other file operations, it retains the `fs` capability only indirectly. That's why `fs` does not show up in the capture set of `xs`.

Capturing types come with a subtype relation where types with "smaller" capture sets are subtypes of types with larger sets (the *subcapturing* relation is defined in more detail below). If a type `T` does not have a capture set, it is called *pure*, and is a subtype of any capturing type that adds a capture set to `T`.

2.2 Function Types

The function type `A => B` stands for a function that can capture arbitrary capabilities. We call such functions *impure*. By contrast, the new single arrow function type `A -> B` stands for a function that cannot capture any capabilities, or otherwise said, is *pure*. One can add a capture set in front of an otherwise pure function. For instance, `{c, d} A -> B` would be a function that can capture capabilities `c` and `d`, but no others.

The impure function type `A => B` is treated as an alias for `{*} A -> B`. That is, impure functions are functions that can capture anything.

Function types and captures both associate to the right, so `{c} A -> {d} B -> C` is the same as `{c} (A -> {d} (B -> C))`.

Contrast with `({c} A) -> ({d} B) -> C` which is a curried pure function over argument types that can capture `c` and `d`, respectively.

Note. Like other object-functional languages, Scala distinguishes between functions and methods (which are defined using **`def`**). Functions are values whereas methods represent pieces of code that logically form part of the enclosing object. The type system treats method signatures and function types separately: A function type is treated as an object type with a single `apply` method. Methods are converted to functions by eta expansion, i.e. the unapplied method reference `m` is transparently converted to the function value `x => m(x)`.

Since methods are not values in Scala, they never capture anything directly. Therefore, the distinctions between pure vs impure function types do not apply to methods. The capabilities captured by a method would show up in the object closure of which the method forms part.

2.3 Capture Checking of Closures

If a closure refers to capabilities in its body, it captures these capabilities in its type. For instance, consider:

```
def test(fs: FileSystem): {fs} String -> Unit =
  (x: String) => Logger(fs).log(x)
```

Here, the body of `test` is a lambda that refers to the capability `fs`, which means that `fs` is retained in the lambda. Consequently, the type of the lambda is `{fs} String -> Unit`.

Note. On the term level, function values are always written with => (or ?=> for context functions). There is no syntactic distinction for pure *vs* impure function values. The distinction is only made in their types.

A closure also captures all capabilities that are captured by the functions it calls. For instance, in

```
def test(fs: FileSystem) =
  def f() = (x: String) => Logger(fs).log(x)
  def g() = f()
  g
```

the result of `test` has type `{fs} String -> Unit` even though function `g` itself does not refer to `fs`.

2.4 Subtyping and Subcapturing

Capturing influences subtyping. As usual we write $T_1 <: T_2$ to express that the type T_1 is a subtype of the type T_2, or equivalently, that T_1 conforms to T_2. An analogous *subcapturing* relation applies to capture sets. If C_1 and C_2 are capture sets, we write $C_1 <: C_2$ to express that C_1 *is covered by* C_2, or, swapping the operands, that C_2 *covers* C_1.

Subtyping extends as follows to capturing types:

- Pure types are subtypes of capturing types. That is, $T <: C\,T$, for any type T, capturing set C.
- For capturing types, smaller capture sets produce subtypes: $C_1\,T_1 <: C_2\,T_2$ if $C_1 <: C_2$ and $T_1 <: T_2$.

A subcapturing relation $C_1 <: C_2$ holds if C_2 *accounts for* every element c in C_1. This means one of the following two conditions must be true:

- $c \in C_2$,
- c's type has capturing set C and C_2 accounts for every element of C (that is, $C <: C_2$).

Example. Given

```
fs: {*} FileSystem
ct: {*} CanThrow[Exception]
l : {fs} Logger
```

we have

```
{l}  <: {fs}     <: {*}
{fs} <: {fs, ct} <: {*}
{ct} <: {fs, ct} <: {*}
```

The set consisting of the root capability `{*}` covers every other capture set. This is a consequence of the fact that, ultimately, every capability is created from `*`.

2.5 Capture Tunneling

Next, we discuss how type-polymorphism interacts with reasoning about capture. To this end, consider the following simple definition of a `Pair` class:

```
class Pair[+A, +B](x: A, y: B):
  def fst: A = x
  def snd: B = y
```

What happens if we pass arguments to the constructor of `Pair` that capture capabilities?

```
def x: {ct} Int -> String
def y: {fs} Logger
def p = Pair(x, y)
```

Here the arguments `x` and `y` close over different capabilities `ct` and `fs`, which are assumed to be in scope. So what should the type of `p` be? Maybe surprisingly, it will be typed as:

```
def p: {} Pair[{ct} Int -> String, {fs} Logger] = Pair(x, y)
```

That is, the outer capture set is empty and does neither mention `ct` nor `fs`, even though the value `Pair(x, y)` *does* capture them. So why don't they show up in its type at the outside?

While assigning `p` the capture set `{ct, fs}` would be sound, types would quickly grow inaccurate and unbearably verbose. To remedy this, $\text{CC}_{<:\Box}$ performs *capture tunneling*. Once a type variable is instantiated to a capturing type, the capture is not propagated beyond this point. On the other hand, if the type variable is instantiated again on access, the capture information "pops out" again.

Even though `p` is technically untracked because its capture set is empty, writing `p.fst` would record a reference to the captured capability `ct`. So if this access was put in a closure, the capability would again form part of the outer capture set. *E.g.*,

```
() => p.fst : {ct} () -> {ct} Int -> String
```

In other words, references to capabilities "tunnel through" generic instantiations—from creation to access; they do not affect the capture set of the enclosing generic data constructor applications. As mentioned above, this principle plays an important part in making capture checking concise and practical. To illustrate, let us take a look at the following example:

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

Relying on capture tunneling, neither the types of the parameters to `mapFirst`, nor its result type need to be annotated with capture sets. Intuitively, the capture sets do not matter for `mapFirst`, since parametricity forbids it from inspecting the actual values inside the pairs. If not for capture tunneling, we would need to annotate `p` as `{*} Pair[A,B]`, since both `A` and `B` and through them, `p` can capture arbitrary capabilities. In turn, this means that for the same reason, without tunneling we would also have `{*} Pair[C,B]` as the result type. This is of course unacceptably inaccurate.

Section 3 describes the foundational theory on which capture checking is based. It makes tunneling explicit through so-called *box* and *unbox* operations. Boxing hides a capture set and unboxing recovers it. The capture checker inserts virtual box and unbox operations based on actual and expected types similar to the way the type checker inserts implicit conversions. Boxing and unboxing has no runtime effect, so the insertion of these operations is only simulated, but not kept in the generated code.

2.6 Escape Checking

Following the principle of object capabilities, the universal capability `*` should conceptually only be available as a parameter to the main program. Indeed, if it was available everywhere, capability checking would be undermined since one could mint new capabilities at will. In line with this reasoning, some capture sets are restricted and must not contain the universal capability.

Specifically, if a capturing type is an instance of a type variable, that capturing type is not allowed to carry the universal capability `{*}`.[2] There is a connection to tunneling here. The capture set of a type has to be present in the environment when a type is instantiated from a type variable. But `*` is not itself available as a global entity in the environment. Hence, this should result in an error.

Using this principle, we can show why the introductory example in Section 1 reported an error. To recall, function `usingFile` was declared like this:

```
def usingFile[T](name: String, op: ({*} FileOutputStream) => T): T = ...
```

The capture checker rejects the illegal definition of `later`

```
val later = usingFile("out",
        f => (y: Int) => xs.foreach(x => f.write(x + y)))
```

with the following error message

```
|    val later = usingFile("out", f => (y: Int) => xs.foreach(x => f.write(x + y)))
|                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
|The expression's type {*} Int -> Unit is not allowed to capture the root capability `*`
|This usually means that a capability persists longer than its allowed lifetime.
```

This error message was produced by the following reasoning steps:

- Parameter `f` has type `{*} FileOutputStream`, which makes it a capability.
- Therefore, the type of the expression

  ```
  (y: Int) => xs.foreach(x => f.write(x + y))
  ```

  is `{f} Int -> Unit`.
- Consequently, we assign the whole closure passed to `usingFile` the dependent function type `(f: {*} FileOutputStream) -> {f} Int -> Unit`.
- The expected type of the closure is a simple, parametric, impure function type `({*} FileOutputStream) => T`, for some instantiation of the type variable `T`.
- We cannot instantiate `T` with `{f} Int -> Unit` since the expected function type is non-dependent. The smallest supertype that matches the expected type is thus `({*} FileOutputStream) => {*} Int -> Unit`.
- Hence, the type variable `T` is instantiated to `{*} Int -> Unit`, which is not allowed and causes the error.

2.7 Escape Checking of Mutable Variables

Another way one could try to undermine capture checking would be to assign a closure with a local capability to a global variable. For instance like this[3]:

```
var loophole: {*} () -> Unit = () => ()
usingFile("tryEscape", f =>
  loophole = () => f.write(0)
```

[2]This follows since type variables range over pure types, so `*` must appear under a box. But rule (Box) in Figure 3 restricts variables in boxed capture sets to be declared in the enclosing environment, which does not hold for `*`.

[3]Mutable variables are not covered by the formal treatment of $CC_{<:\Box}$. We include the discussion anyway to show that escape checking can be generalized to scope extrusions separate from result values.

```
}
loophole()
```

We prevent such scope extrusions by imposing the restriction that mutable variables cannot have types with universal capture sets.

One also needs to prevent returning or assigning a closure with a local capability in an argument of a parametric type. For instance, here is a slightly more refined attack:

```
val sneaky = usingLogFile { f => Pair(() => f.write(0), 1) }
sneaky.fst()
```

At the point where the `Pair` is created, the capture set of the first argument is `{f}`, which is OK. But at the point of use, it is `{*}`: since `f` is no longer in scope we need to widen the type to a supertype that does not mention it (*c.f.* the explanation of avoidance in Section 3.3). This causes an error, again, as the universal capability is not permitted to be in the unboxed form of the return type (*c.f.* the precondition of (Unbox) in Figure 3).

Variable	x, y, z		
Type Variable	X, Y, Z		
Value	v, w	::=	$\lambda(x : T)\, t \quad \mid \quad \lambda[X <: S]t \quad \mid \quad \square\, x$
Answer	a	::=	$v \quad \mid \quad x$
Term	s, t	::=	$a \quad \mid \quad x\, y \quad \mid \quad x\, [S] \quad \mid \quad \mathbf{let}\, x = s\, \mathbf{in}\, t \quad \mid \quad C \multimap x$
Shape Type	S	::=	$X \quad \mid \quad \top \quad \mid \quad \forall(x : U)T \quad \mid \quad \forall[X <: S]T \quad \mid \quad \square\, T$
Type	T, U	::=	$S \quad \mid \quad C\, S$
Capture Set	C	::=	$\{x_1, \ldots, x_n\}$

Fig. 2. Syntax of System $\mathsf{CC}_{<:\square}$

3 THE $\mathsf{CC}_{<:\square}$ CALCULUS

The syntax of $\mathsf{CC}_{<:\square}$ is given in Figure 2. In short, it describes a dependently typed variant of System $\mathsf{F}_{<:}$ in monadic normal form (MNF) with capturing types and boxes.

Dependently typed: Types may refer to term variables in their capture sets, which introduces a simple form of (variable-)dependent typing. As a consequence, a function's result type may now refer to the parameter in its capture set. To be able to express this, the general form of a function type $\forall(x : U)T$ explicitly names the parameter x. We retain the non-dependent syntax $U \to T$ for function types as an abbreviation if the parameter is not mentioned in the result type T.

Dependent typing is attractive since it means that we can refer to object capabilities directly in types, instead of having to go through auxiliary region or effect variables. We thus avoid clutter related to quantification of such auxiliary variables.

Monadic normal form: The term structure of $\mathsf{CC}_{<:\square}$ requires operands of applications to be variables. This does not constitute a loss of expressiveness, since a general application $t_1\, t_2$ can be expressed as $\mathbf{let}\, x_1 = t_1\, \mathbf{in}\, \mathbf{let}\, x_2 = t_2\, \mathbf{in}\, x_1\, x_2$. This syntactic convention has advantages for variable-dependent typing. In particular, typing function application in such a calculus requires substituting actual arguments for formal parameters. If arguments are restricted to be variables,

these substitutions are just variable/variable renamings, which keep the general structure of a type. If arguments were arbitrary terms, such a substitution would in general map a type to something that was not syntactically a type. Monadic normal form [Hatcliff and Danvy 1994] is a slight generalization of the better-known A-normal form (ANF) [Sabry and Felleisen 1993] to allow arbitrary nesting of let expressions. We use a here a variant of MNF where applications are over variables instead of values.

A similar restriction to MNF was employed in DOT [Amin et al. 2016], the foundation of Scala's object model, for the same reasons. The restriction is invisible to source programs, which can still be in direct style. For instance, the Scala compiler selectively translates a source expression in direct style to MNF if a non-variable argument is passed to a dependent function. Type checking then takes place on the translated version.

Capturing Types: The types in $\mathsf{CC}_{<:\Box}$ are stratified as *shape types* S and regular types T. Regular types can be shape types or capturing types $\{x_1, \ldots, x_n\}\ S$. Shape types are made up from the usual type constructors in $\mathsf{F}_{<:}$ plus boxes. We freely use shape types in place of types, assuming the equivalence $\{\}\ S \equiv S$.

Boxes: Type variables X can be bounded or instantiated only with shape types, not with regular types. To make up for this restriction, a regular type T can be encapsulated in a shape type by prefixing it with a box operator $\Box\ T$. On the term level, $\Box\ x$ injects a variable into a boxed type. A variable of boxed type is unboxed using the syntax $C\ \multimap\ x$ where C is the capture set of the underlying type of x. We have seen in Section 2 that boxing and unboxing allow a kind of capability tunneling by omitting capabilities when values of parametric types are constructed and charging these capabilities instead at use sites.

System $\mathsf{F}_{<:}$: We base $\mathsf{CC}_{<:\Box}$ on a standard type system that supports the two principal forms of polymorphism, subtyping and universal.

Subtyping comes naturally with capabilities in capture sets. First, a type capturing fewer capabilities is naturally a subtype of a type capturing more capabilities, and pure types are naturally subtypes of capturing types. Second, if capability x is derived from capability y, then a type capturing x can be seen as a subtype of the same type but capturing y.

Universal polymorphism poses specific challenges when capture sets are introduced which are addressed in $\mathsf{CC}_{<:\Box}$ by the stratification into shape types and regular types and the box/unbox operations that map between them.

Note that the only form of term dependencies in $\mathsf{CC}_{<:\Box}$ relate to capture sets in types. If we omit capture sets and boxes, the calculus is equivalent to standard $\mathsf{F}_{<:}$, despite the different syntax. We highlight in the figures the essential additions wrt $\mathsf{F}_{<:}$ with a grey background.

$\mathsf{CC}_{<:\Box}$ is intentionally meant to be a small and canonical core calculus that does not cover higher-level features such as records, modules, objects, or classes. While these features are certainly important, their specific details are also somewhat more varied and arbitrary than the core that's covered. Many different systems can be built on $\mathsf{CC}_{<:\Box}$, extending it with various constructs to organize code and data on higher-levels.

Capture Sets. Capture sets C are finite sets of variables of the form $\{x_1, \ldots, x_n\}$. We understand $*$ to be a special variable that can appear in capture sets, but cannot be bound in Γ. We write $C \setminus x$ as a shorthand for $C \setminus \{x\}$.

Capture sets of closures are determined using a function cv over terms.

Definition (Captured Variables). The captured variables $\mathrm{cv}(t)$ of a term t are given as follows.

$$
\begin{array}{lcll}
\mathrm{cv}(\lambda(x : T)t) & = & \mathrm{cv}(t)\backslash x & \\
\mathrm{cv}(\lambda[X <: S]t) & = & \mathrm{cv}(t) & \\
\mathrm{cv}(x) & = & \{x\} & \\
\mathrm{cv}(\textbf{\textit{let}}\, x = v\, \textbf{\textit{in}}\, t) & = & \mathrm{cv}(t) & \textbf{\textit{if}}\, x \notin \mathrm{cv}(t) \\
\mathrm{cv}(\textbf{\textit{let}}\, x = s\, \textbf{\textit{in}}\, t) & = & \mathrm{cv}(s) \cup \mathrm{cv}(t)\backslash x & \\
\mathrm{cv}(x\, y) & = & \{x, y\} & \\
\mathrm{cv}(x[S]) & = & \{x\} & \\
\mathrm{cv}(\Box\, x) & = & \{\} & \\
\mathrm{cv}(C \multimap x) & = & C \cup \{x\} &
\end{array}
$$

The definitions of captured and free variables of a term are very similar, with the following three differences:

(1) Boxing a term $\Box\, x$ obscures x as a captured variable.
(2) Dually, unboxing a term $C \multimap x$ counts the variables in C as captured.
(3) In an evaluated let binding $\textbf{let}\, x = v\, \textbf{in}\, t$, the captured variables of v are counted only if x is a captured variable of t.

The first two rules encapsulate the essence of box-unbox pairs: Boxing a term obscures its captured variable and makes it necessary to unbox the term before its value can be accessed; unboxing a term presents variables that were obscured when boxing.

Figure 3 presents typing and evaluation rules for $\mathrm{CC}_{<:\Box}$. There are four main sections on subcapturing, subtyping, typing, and evaluation. These are explained in the following.

3.1 Subcapturing

Subcapturing establishes a preorder relation on capture sets that gets propagated to types. Smaller capture sets with respect to subcapturing lead to smaller types with respect to subtyping.

The relation is defined by three rules. The first two rules (sc-set) and (sc-elem) establish that subsets imply subcaptures. That is, smaller capture sets subcapture larger ones. The last rule (sc-var) is the most interesting since it reflects an essential property of object capabilities. It states that a variable x of capturing type $C\ S$ generates a capture set $\{x\}$ that subcaptures the capabilities C with which the variable was declared. In a sense, (sc-var) states a monotonicity property: a capability refines the capabilities from which it is created. In particular, capabilities cannot be created from nothing. Every capability needs to be derived from some more sweeping capabilities which it captures in its type.

The rule also validates our definition of capabilities as variables with non-empty capture sets in their types. Indeed, if a variable is defined as $x : \{}\ S$, then by (sc-var) we have $\{x\} <: \{\}$. This means that the variable can be disregarded in the formation of cv, for instance. Even if x occurs in a term, a capture set with x in it is equivalent (with respect to mutual subcapturing) to a capture set without. Hence, x can safely be dropped without affecting subtyping or typing.

Rules (sc-set) and (sc-elem) mean that if set C is a subset of C', we also have $C <: C'$. But the reverse is not true. For instance, with (sc-var) we can derive the following relationship assuming lambda-bound variables x and y:

$$x : \{*\}\ \top, y : \{x\}\ \top \vdash \{y\} <: \{x\}$$

Intuitively this makes sense, as y can capture no more than x. However, we *cannot* derive $\{x\} <: \{y\}$, since arguments passed for y may in fact capture *less* than x, e.g. they could be pure.

While there are no subcapturing rules for top or bottom capture sets, we can still establish:

Subcapturing $\boxed{\Gamma \vdash C <: C}$

$$\frac{x \in C}{\Gamma \vdash \{x\} <: C}\ (\textsc{sc-elem}) \qquad \frac{\Gamma \vdash \{x_1\} <: C \ \ldots\ \Gamma \vdash \{x_n\} <: C}{\Gamma \vdash \{x_1, \ldots, x_n\} <: C}\ (\textsc{sc-set})$$

$$\frac{x : C'\ S \in \Gamma \quad \Gamma \vdash C' <: C}{\Gamma \vdash \{x\} <: C}\ (\textsc{sc-var})$$

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

$$\Gamma \vdash T <: T\ (\textsc{refl}) \qquad \frac{\Gamma \vdash T_1 <: T_2 \quad \Gamma \vdash T_2 <: T_3 \quad \Gamma \vdash T_2\ \mathbf{wf}}{\Gamma \vdash T_1 <: T_3}\ (\textsc{trans})$$

$$\frac{X <: S \in \Gamma}{\Gamma \vdash X <: S}\ (\textsc{tvar}) \qquad \Gamma \vdash S <: \top\ (\textsc{top})$$

$$\frac{\Gamma \vdash U_2 <: U_1 \qquad \Gamma, x : U_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x : U_1)T_1 <: \forall(x : U_2)T_2}\ (\textsc{fun}) \qquad \frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, X <: S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall[X <: S_1]T_1 <: \forall[X <: S_2]T_2}\ (\textsc{tfun})$$

$$\frac{\Gamma \vdash C_1 <: C_2 \qquad \Gamma \vdash S_1 <: S_2}{\Gamma \vdash C_1\ S_1 <: C_2\ S_2}\ (\textsc{capt}) \qquad \frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash \Box\, T_1 <: \Box\, T_2}\ (\textsc{boxed})$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x : C\ S \in \Gamma}{\Gamma \vdash x : \{x\}\ S}\ (\textsc{var}) \qquad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash T <: U \qquad \Gamma \vdash U\ \mathbf{wf}}{\Gamma \vdash t : U}\ (\textsc{sub})$$

$$\frac{\Gamma, x : U \vdash t : T \qquad \Gamma \vdash U\ \mathbf{wf}}{\Gamma \vdash \lambda(x : U)t : \mathrm{cv}(t)\backslash x\ \forall(x : U)T}\ (\textsc{abs}) \qquad \frac{\Gamma, X <: S \vdash t : T \qquad \Gamma \vdash S\ \mathbf{wf}}{\Gamma \vdash \lambda[X <: S]t : \mathrm{cv}(t)\ \forall[X <: S]T}\ (\textsc{tabs})$$

$$\frac{\Gamma \vdash x : C\ \forall(z : U)T \qquad \Gamma \vdash y : U}{\Gamma \vdash x\, y : [z := y]T}\ (\textsc{app}) \qquad \frac{\Gamma \vdash x : C\ \forall[X <: S]T}{\Gamma \vdash x\,[S] : [X := S]T}\ (\textsc{tapp})$$

$$\frac{\Gamma \vdash x : C\ S \qquad C \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash \Box\, x : \Box\, C\ S}\ (\textsc{box}) \qquad \frac{\Gamma \vdash x : \Box\, C\ S \qquad C \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash C \multimap x : C\ S}\ (\textsc{unbox})$$

$$\frac{\Gamma \vdash s : T \qquad \Gamma, x : T \vdash t : U \qquad x \notin \mathrm{fv}(U)}{\Gamma \vdash \mathbf{let}\, x = s\, \mathbf{in}\, t : U}\ (\textsc{let})$$

Evaluation $\boxed{\Gamma \vdash t \longrightarrow t'}$

$$\begin{array}{lcll}
\sigma[\, e[\, x\, y\,]\,] & \longrightarrow & \sigma[\, e[\, [z := y]t\,]\,] & \mathbf{if}\ \sigma(x) = \lambda(z : T)t \quad (\textsc{apply}) \\
\sigma[\, e[\, x\,[S]\,]\,] & \longrightarrow & \sigma[\, e[\, [X := S]t\,]\,] & \mathbf{if}\ \sigma(x) = \lambda[X <: S']t \quad (\textsc{tapply}) \\
\sigma[\, e[\, C \multimap x\,]\,] & \longrightarrow & \sigma[\, e[\, y\,]\,] & \mathbf{if}\ \sigma(x) = \Box\, y \quad (\textsc{open}) \\
\sigma[\, e[\, \mathbf{let}\, x = y\, \mathbf{in}\, t\,]\,] & \longrightarrow & \sigma[\, e[\, [x := y]t\,]\,] & \quad (\textsc{rename}) \\
\sigma[\, e[\, \mathbf{let}\, x = v\, \mathbf{in}\, t\,]\,] & \longrightarrow & \sigma[\, \mathbf{let}\, x = v\, \mathbf{in}\, e[\, t\,]\,] & \mathbf{if}\ e \neq [\,] \quad (\textsc{lift})
\end{array}$$

$$\begin{array}{llll}
\mathbf{where} & \mathbf{Store\ context} & \sigma & ::= \quad [\,] \ \mid\ \mathbf{let}\, x = v\, \mathbf{in}\, \sigma \\
 & \mathbf{Eval\ context} & e & ::= \quad [\,] \ \mid\ \mathbf{let}\, x = e\, \mathbf{in}\, t
\end{array}$$

Fig. 3. Typing and Evaluation Rules of System $\mathrm{CC}_{<:\Box}$

Proposition 3.1. *If C is well-formed in Γ, then $\Gamma \vdash \{\} <: C <: \{*\}$.*

A proof is enclosed in the appendix.

Proposition 3.2. *The subcapturing relation $\Gamma \vdash \_ <: \_$ is a preorder.*

Proof. We can show that transitivity and reflexivity are admissible. □

3.2 Subtyping

The subtyping rules of $\mathsf{CC}_{<:\Box}$ are very similar to those of System $\mathsf{F}_{<:}$, with the only significant addition being the rules for capturing and boxed types. Note that as $S \equiv \{\}\ S$, both transitivity and reflexivity apply to shape types as well. (CAPT) allows comparing types that have capture sets, where smaller capture sets lead to smaller types. (BOXED) propagates subtyping relations between types to their boxed versions.

3.3 Typing

Typing rules are again close to System $\mathsf{F}_{<:}$, with differences to account for capture sets.

Rule (VAR) is the basis for capability refinements. If x is declared with type $C\ S$, then the type of x has $\{x\}$ as its capture set instead of C. The capturing set $\{x\}$ is more specific than C, in the subcapturing sense. Therefore, we can recover the capture set C through subsumption.

Rules (ABS) and (TABS) augment the abstraction's type with a capture set that contains the captured variables of the term. Through subsumption and rule (SC-VAR), untracked variables can immediately be removed from this set.

The (APP) rule substitutes references to the function parameter with the argument to the function. This is possible since arguments are guaranteed to be variables. The function's capture set C is disregarded, reflecting the principle that the function closure is consumed by the application. Rule (TAPP) is analogous.

Aside: A more conventional version of (TAPP) would be

$$\frac{\Gamma \vdash x : C\ \forall[X <: S']T \qquad \Gamma \vdash S <: S'}{\Gamma \vdash x[S] : [X := S]T} \tag{TAPP'}$$

That formulation is equivalent to (TAPP) in the sense that either rule is derivable from the other, using subsumption and contravariance of type bounds.

Rules (BOX) and (UNBOX) map between boxed and unboxed types. They require all members of the capture set under the box to be bound in the environment Γ. Consequently, while one can create a boxed type with $\{*\}$ as its capture set through subsumption, one cannot unbox values of this type. This property is fundamental for ensuring scoping of capabilities.

Avoidance. As is usual in dependent type systems, Rule (LET) has as a side condition that the bound variable x does not appear free in the result type U. This so called *avoidance* property is usually attained through subsumption. For instance consider an enclosing capability $c : T_1$ and the term

$$\textbf{let}\ x = \lambda(y : T_2).c\ \textbf{in}\ \lambda(z : \{x\}\ T_3).z$$

The most specific type of x is $\{c\}\ (T_2 \to T_1)$ and the most specific type of the body of the let is $\forall(z : \{x\}\ T_3).\{z\}\ T_3$. We need to find a supertype of the latter type that does not mention x. It turns out the most specific such type is $T_3 \to \{c\}T_3$, so that is a possible type of the let, and it should be the inferred type.

In general there is always a most specific avoiding type for a (LET):

Proposition 3.3. *Consider a term* $\textbf{let}\, x = s\, \textbf{in}\, t$ *in an environment* Γ *such that* $\Gamma \vdash s : T_1$ *and* $\Gamma, x : T_1 \vdash t : T_2$. *Then there exists a minimal (wrt* $<:$*) type* T_3 *such that* $T_2 <: T_3$ *and* $x \notin \mathrm{fv}(T_3)$

Proof. Without loss of generality, assume that $\Gamma \vdash s : C_s\ U$ is the most specific typing for s in Γ and $\Gamma, x : C_s\ U \vdash t : T$ is the most specific typing for t in the context of the body of the let, namely $\Gamma, x : C_s\ U$. Let T' be constructed from T by replacing x with C_s in covariant capture set positions and by replacing x with the empty set in contravariant capture set positions. Then for every type V avoiding x such that $\Gamma, x : C_s\ S \vdash T <: V, \Gamma \vdash T' <: V$. This is shown by a straightforward structual induction, which we give in the appendix in Section A.61. □

3.4 Well-formedness

Well-formedness $\Gamma \vdash T\ \mathbf{wf}$ is equivalent to well-formedness in System $\mathsf{F}_{<:}$ in that free variables in types and terms must be defined in the environment, except that capturing types may mention the universal capability $*$ in their capture sets:

$$\frac{\Gamma \vdash S\ \mathbf{wf} \qquad C \subseteq \mathrm{dom}(\Gamma) \cup \{*\}}{\Gamma \vdash C\ S\ \mathbf{wf}} \tag{CAPT-WF}$$

3.5 Evaluation

Evaluation is defined by a small-step reduction relation. This relation is quite different from usual reduction via term substitution. Substituting values for variables would break the monadic normal form of a program. Instead, we reduce the right hand sides of let-bound variables in place and lookup the bindings in the environment of a redex.

Every redex is embedded in an outer *store context* and an inner *evaluation context*. These represent orthogonal decompositions of let bindings. An evaluation context e always puts the focus [] on the right-hand side t_1 of a let binding $\textbf{let}\, x = t_1\, \textbf{in}\, t_2$. By contrast, a store context σ puts the focus on the following term t_2 and requires that t_1 is evaluated.

The first three rules — (APPLY), (TAPPLY), (OPEN) — rewrite simple redexes: applications, type applications and unboxings. Each of these rules looks up a variable in the enclosing store and proceeds based on the value that was found.

The last two rules are administrative in nature. They both deal with evaluated **let**s in redex position. If the right hand side of the **let** is a variable, the **let** gets expanded out by renaming the bound variable using (RENAME). If it is a value, the **let** gets lifted out into the store context using (LIFT).

Proposition 3.4. *Evaluation is deterministic. For any term* t_1 *there is at most one term* t_2 *such that* $t_1 \longrightarrow t_2$.

Proof. By a straightforward inspection of the reduction rules and definitions of contexts.

4 METATHEORY

We prove that $\mathrm{CC}_{<:\Box}$ is sound through the standard progress and preservation theorems. The proofs for all the lemmas and theorems stated in this section are provided in the appendix.

In order to prove both progress and preservation, we need technical lemmas that allow manipulation of typing judgements for terms under store and evaluation contexts. To state these lemmas, we first need to define what it means for typing and store contexts to match, which we do in Figure 4.

$$\frac{\Gamma, x : T \vdash \sigma \sim \Delta \qquad \Gamma \vdash v : T \qquad x \notin \mathrm{fv}(T)}{\Gamma \vdash \mathbf{let}\, x = v\, \mathbf{in}\, \sigma \sim x : T, \Delta} \qquad \Gamma \vdash [\,] \sim \cdot$$

Fig. 4. Matching environment $\boxed{\Gamma \vdash \sigma \sim \Delta}$

Having $\Gamma \vdash \sigma \sim \Delta$ lets us know that σ is well-typed in Γ if we use Δ as the types of the bindings. Using this definition, we can state the following four lemmas, which also illustrate how the store and evaluation contexts interact with typing:

Definition 4.1 (Evaluation context typing ($\Gamma \vdash e : U \Rightarrow T$)). We say that Γ types an evaluation context e as $U \Rightarrow T$ if $\Gamma, x : U \vdash e[\, x \,] : T$.

Lemma 4.2 (Evaluation context typing inversion).
$\Gamma \vdash e[\, s \,] : T$ implies that for some U we have $\Gamma \vdash e : U \Rightarrow T$ and $\Gamma \vdash s : U$.

Lemma 4.3 (Evaluation context reification).
If both $\Gamma \vdash e : U \Rightarrow T$ and $\Gamma \vdash s : U$, then $\Gamma \vdash e[\, s \,] : T$.

Lemma 4.4 (Store context typing inversion).
$\Gamma \vdash \sigma[\, t \,] : T$ implies that for some Δ we have $\Gamma \vdash \sigma \sim \Delta$ and $\Gamma, \Delta \vdash t : T$.

Lemma 4.5 (Store context reification).
If both $\Gamma, \Delta \vdash t : T$ and $\Gamma \vdash \sigma \sim \Delta$, then also $\Gamma \vdash \sigma[\, t \,] : T$.

We can now proceed to our main theorems; their statements differ slightly from System $\mathsf{F}_{<:}$, as we need to account for our monadic normal form. Our preservation theorem captures that the important type to preserve is the one assigned to the term under the store. It is stated as follows:

Theorem 4.6 (Preservation). *If we have $\Gamma \vdash \sigma \sim \Delta$ and $\Gamma, \Delta \vdash t : T$, then $\sigma[\, t \,] \longrightarrow \sigma[\, t' \,]$ implies that $\Gamma, \Delta \vdash t' : T$.*

To neatly state the progress theorem, we first need to define canonical store-plug splits, which simply formally express that the store context wrapping a term is as large as possible:

Definition 4.7 (Canonical store-plug split). We say that a term of the form $\sigma[\, t \,]$ is a *canonical split* (of the entire term into store context σ and the plug t) if t is not of the form $\mathbf{let}\, x : T = v\, \mathbf{in}\, t'$.

With this definition, we can now state the progress theorem:

Theorem 4.8 (Progress). *If $\vdash \sigma[\, e[\, t \,] \,] : T$ and $\sigma[\, e[\, t \,] \,]$ is a canonical store-plug split, then either $e[\, t \,] = a$, or there exists $\sigma[\, t' \,]$ such that $\sigma[\, e[\, t \,] \,] \longrightarrow \sigma[\, t' \,]$.*

The lemmas needed to prove progress and preservation are for the most part standard. As our calculus is term-dependent, we also need to account for term substitution affecting both environments and types, not only terms. For instance, the lemma stating that term substitution preserves typing is expressed as follows:

Lemma 4.9 (Term substitution preserves typing).
If $\Gamma, x : U, \Delta \vdash t : T$ and $\Gamma \vdash y : U$, then $\Gamma, [x := y]\Delta \vdash [x := y]t : [x := y]T$.

In this statement, we can also see that we only consider substituting one term variable for another, due to MNF. Using MNF affects other parts of the proof as well – in addition to typical canonical forms lemmas, we also need to show that looking up the value bound to a variable in a store preserves the types we can assign to the variable:

Lemma 4.10 (Variable lookup inversion).
If we have both $\Gamma \vdash \sigma \sim \Delta$ and $x : C\, R \in \Gamma, \Delta$, then $\sigma(x) = v$ implies that $\Gamma, \Delta \vdash v : C\, R$.

Capture sets and captured variables. Our typing rules use cv to calculate the capture set that should be assigned to terms. With that in mind, we can ask the question: what is the exact relationship between captured variables and capture sets we use to type the terms?

Because of subcapturing, this relationship is not as obvious as it might seem. For fully evaluated terms (of the form $\sigma[a]$), their captured variables are the most precise capture set they can be assigned. The following lemma states this formally:

Lemma 4.11 (Capture prediction for answers). *If* $\Gamma \vdash \sigma[a] : C\ S$, *then* $\Gamma \vdash \mathrm{cv}(\sigma[a]) <: C$.

If we start with an unreduced term $\sigma[t]$, then the situation becomes more complex. It can mention and use capabilities that will not be reflected in the capture set at all – for instance, if $t = x\ y$, the capture set of x is irrelevant to the type assigned to t by (app). However, if $\sigma[t]$ reduces fully to a term of the form $\sigma[\sigma'[a]]$, the captured variables of $\sigma'[a]$ will correspond to capture sets we could assign to t.

In other words, the capture sets we assign to unevaluated terms under a store context predict variables that will be captured by the answer those terms reduce to. Formally we can express this as follows:

Lemma 4.12 (Capture prediction for terms).
Let $\vdash \sigma \sim \Delta$ *and* $\Delta \vdash t : C\ S$. *Then* $\sigma[t] \longrightarrow^* \sigma[\sigma'[a]]$ *implies that* $\Delta \vdash \mathrm{cv}(\sigma'[a]) <: C$.

4.1 Correctness of boxing

Boxing capabilities allows temporarily hiding them from capture sets. Given that we can do so, one may be inclined to ask: what is the correctness criterion for our typing rules for box and unbox forms?

Intuitively, we should not be able to "smuggle in" a capability by using boxes: all the capabilities of a term should be somehow accounted for. A basic criterion our typing rules should satisfy is that a term that boxes and immediately unboxes a capability should have a cv at least as wide than that of the capability, i.e. a cv that accounts for the capability. Formally we can state this property as follows:

Proposition 4.13. *Let* $\vdash \sigma \vdash \Delta$ *and let* $t = \mathbf{let}\ y = \square\ x\ \mathbf{in}\ C \multimap y$ *such that for some* e, T, *we have* $\Delta \vdash \sigma[\ e[\ t\ ]\ ] : T$. *Then we also have:*

$$\Delta \vdash \{x\} <: C = \mathrm{cv}(t)$$

It is straightforward to verify that this holds based on the progress theorem and by induction on the typing derivation. Ideally, our correctness criterion would speak about arbitrary terms, by characterising their captured variables alongside reduction paths. Since this clearly doesn't make sense for closed terms, we will need one additional definition: a way of introducing well-behaved capabilities. A *platform* Ψ is an outer portion of the store which formally represents primitive capabilities. It is defined as follows:

$$\begin{array}{rcll} \Psi & ::= & \mathbf{let}\ x = v\ \mathbf{in}\ \Psi & \text{if } \mathrm{fv}(v) = \{\} \\ & | & [\,] & \end{array}$$

Since a platform Ψ is meant to introduce primitive capabilities, it doesn't make sense to type them with any other capture set than $\{*\}$. The following definition captures this idea:

Definition (Well-typed program). *We say that a term* $\Psi[\ t\ ]$ *is a well-typed program if for some typing context* Δ *such that* $\vdash \Psi \sim \Delta$ *we have* $\Delta \vdash t : T$, *and for all* $x \in \mathrm{dom}(\Psi)$ *we have* $x : \{*\}\ S \in \Delta$.

The capabilities introduced by the platform of a well-typed program are well-behaved in the sense that their values cannot reference any other variable, and their capture sets are $\{*\}$. Given cv and platform, we can state our desired lemma as follows:

Lemma 4.14 (Program authority preservation). *Let* $\Psi[\ t\ ]$ *be a well-typed program such that* $\Psi[\ t\ ] \longrightarrow \Psi[\ t'\ ]$. *Then* $\mathrm{cv}(t')$ *is a subset of* $\mathrm{cv}(t)$.

The name of this lemma hints at the property we want to demonstrate next, which offers another perspective at what it means for boxes to be correctly typed. This property will capture the connection between the cv of a term and the capabilities it may use during evaluation. We make intuitive usage of this connection when we say, for instance, that a function f typed as $\{\}\ \mathsf{Unit} \to \mathsf{Unit}$ cannot perform any effects when called. Formally, it cannot do so because when we reduce a term of the form $f\ x$, based on (abs) we know it will be replaced by some term such that $\mathrm{cv}(t) = \{\}$. Here is where we make an intuitive jump: since $\mathrm{cv}(t)$ is empty and we do not pass anything to t, it should not use any capabilities during evaluation.

To state this formally, first we need to define which capabilities are used during reduction. All formal capabilities are abstractions, so a natural definition is to say that a capability x was used if it was applied, i.e. if a term of the form $x\ y$ was reduced. To speak about this, we will define $\mathrm{used}(t \longrightarrow s)$ as function from reduction derivations to sets of variables, as follows:

$$
\begin{array}{lcll}
\mathrm{used}(t_1 \longrightarrow t_2 \longrightarrow \cdots \longrightarrow t_n) & = & \mathrm{used}(t_1 \longrightarrow t_2) \cup \mathrm{used}(t_2 \longrightarrow \cdots \longrightarrow t_n) & \\
\mathrm{used}(\sigma[\ e[\ x\ y\ ]\ ] \longrightarrow \sigma[\ t\ ]) & = & \{x\} & \\
\mathrm{used}(\sigma[\ e[\ x\ [T]\ ]\ ] \longrightarrow \sigma[\ t\ ]) & = & \{x\} & \\
\mathrm{used}(t_1 \longrightarrow t_2) & = & \{\} & \text{(otherwise)}
\end{array}
$$

The last case applies to rules (OPEN), (RENAME), (LIFT).

Now, let us consider how exactly the cv of a term is connected to the capabilities it will use during evaluation. If we disregard boxes, then clearly a term t can only use a capability x if it is free in t, which also means that it is in $\mathrm{cv}(t)$. However, boxing a capability hides it from the cv. If we disregard unbox forms, this is again not a problem: a capability under a box cannot be used unless the box is opened first. Now, if we consider unbox forms $C \multimap x$, we encounter the essence of the problem: $\mathrm{cv}(C \multimap x) = C$. On the term level, we have no guarantee that the capture sets of C and x are connected. The (box) and (unbox) typing rules ensure that if a term gains access to a capability by opening a box, then the "key" used to open the box (the C in $C \multimap x$) must account for the boxed capability. Thanks to them, we can say that $\mathrm{cv}(t)$ is the *authority* of the term t: during reduction, t can only use capabilities contained in $\mathrm{cv}(t)$. We formally state this property as follows:

Lemma 4.15 (Used capability prediction). *Let* $\Psi[\ t\ ]$ *be a well-typed program that reduces in zero or more steps to* $\Psi[\ t'\ ]$. *Then the primitive capabilities used during the reduction are a subset of the authority of* t.

5 EXAMPLES

We have implemented a type checker for $\mathrm{CC}_{<:\Box}$ as an extension of the Scala 3 compiler to enable experimentation with larger code examples. Notably, our extension infers which types must be boxed, and automatically generates boxing and unboxing operations when values are passed to and returned from instantiated generic datatypes, so none of these technical details appear in the actual user-written Scala code.

In this section, we present examples that demonstrate the usability of the language. In Section 5.1, we remain close to the core calculus by encoding lists using only functions; here, we still show the boxed types and boxing and unboxing operations that the compiler infers in gray, though they are not in the source code. In Sections 5.2 and 5.3, we use additional Scala features in larger examples,

to implement stack allocation and polymorphic data structures. For these examples, we present the source code without cluttering it with the boxing operations inferred by the compiler.

5.1 Church-Encoded Lists

Using the Scala prototype implementation of $CC_{<:\square}$, the Böhm-Berarducci encoding [Böhm and Berarducci 1985] of a linked list data structure can be implemented and typed as follows. Here, a list is represented by its right fold function:

```
type Op[T <: □ {*} Any, C <: □ {*} Any] =
  (v: T) => (s: C) => C

type List[T <: □ {*} Any] =
  [C <: □ {*} Any] -> (op: Op[T, C]) -> {op} (s: C) -> C

def nil[T <: □ {*} Any]: List[T] =
  [C <: □ {*} Any] => (op: Op[T, C]) => (s: C) => s

def cons[T <: □ {*} Any](hd: T, tl: List[T]): List[T] =
  [C <: □ {*} Any] => (op: Op[T, C]) => (s: C) => op(hd)(tl[C](op)(s))
```

A list inherently captures any capabilities that may be captured by its elements. Therefore, naively, one may expect the capture set of the list to include the capture set of the type `T` of its elements. However, boxing and unboxing enables us to elide the capture set of the elements from the capture set of the containing list. When constructing a list using `cons`, the elements must be boxed:

```
cons(□ 1, cons(□ 2, cons(□ 3, nil)))
```

A `map` function over the list can be implemented and typed as follows:

```
def map[A <: □ {*} Any, B <: □ {*} Any](xs: List[A])(f: {*} A -> B): List[B] =
  {} ○– xs[□ List[B]]((hd: A) => (tl: List[B]) => cons(f(hd), tl))(nil)
```

The mapped function `f` may capture any capabilities, as documented by the capture set `{*}` in its type. However, this does not affect the type of `map` or its result type `List[B]`, since the mapping is strict, so the resulting list does not capture any capabilities captured by `f`. If a value returned by the function `f` were to capture capabilities, this would be reflected in its type, the concrete type substituted for the type variable `B`, and would therefore be reflected in the concrete instantiation of the result type `List[B]` of `map`.

5.2 Stack Allocation

Automatic memory management using a garbage collector is convenient and prevents many errors, but it can impose significant performance overheads in programs that need to allocate large numbers of short-lived objects. If we can bound the lifetimes of some objects to coincide with a static scope, it is much cheaper to allocate those objects on a stack as follows: [4]

```
class Pooled

val stack = mutable.ArrayBuffer[Pooled]()
var nextFree = 0

def withFreshPooled[T](op: Pooled => T): T =
```

[4]For simplicity, this example is neither thread nor exception safe.

```
  if nextFree >= stack.size then stack.append(new Pooled)
  val pooled = stack(nextFree)
  nextFree = nextFree + 1
  val ret = op(pooled)
  nextFree = nextFree - 1
  ret
```

The `withFreshPooled` method calls the provided function `op` with a freshly stack-allocated instance of class `Pooled`. It works as follows. The `stack` maintains a pool of already allocated instances of `Pooled`. The `nextFree` variable records the offset of the first element of `stack` that is available to reuse; elements before it are in use. The `withFreshPooled` method first checks whether the `stack` has any available instances; if not, it adds one to the stack. Then it increments `nextFree` to mark the first available instance as used, calls `op` with the instance, and decrements `nextFree` to mark the instance as freed. In the fast path, allocating and freeing an instance of `Pooled` is reduced to just incrementing and decrementing the integer `nextFree`.

However, this mechanism fails if the instance of `Pooled` outlives the execution of `op`, if `op` captures it in its result. Then the captured instance may still be accessed while at the same time also being reused by later executions of `op`. For example, the following invocation of `withFreshPooled` returns a closure that accesses the `Pooled` instance when it is invoked on the second line, after the `Pooled` instance has been freed:

```
val pooledClosure = withFreshPooled(pooled => () => pooled.toString)
pooledClosure()
```

Using capture sets, we can prevent such captures and ensure the safety of stack allocation just by marking the `Pooled` instance as tracked:

```
def withFreshPooled[T](op: (pooled: {*} Pooled) => T): T =
```

Now the `pooled` instance can be captured only in values whose capture set accounts for `{pooled}`. The type variable `T` cannot be instantiated with such a capture set because `pooled` is not in scope outside of `withFreshPooled`, so only `{*}` would account for `{pooled}`, but we disallowed instantiating a type variable with `{*}`. With this declaration of `withFreshPooled`, the above `pooledClosure` example is correctly rejected, while the following safe example is allowed:

```
withFreshPooled(pooled => pooled.toString)
```

5.3 Collections

In the following examples we show that a typing discipline based on $CC_{<:\Box}$ can be lightweight enough to make capture checking of operations on standard collection types practical. This is important, since such operations are the backbone of many programs. All examples compile with our current capture checking prototype [Scala 2022a].

We contrast the APIs of common operations on Scala's standard collection types `List` and `Iterator` when capture sets are taken into account. Both APIs are expressed as Scala 3 extension methods [Odersky and Martres 2020] over their first parameter. Here is the `List` API:

```
extension [A](xs: List[A])
  def apply(n: Int): A
  def foldLeft[B](z: B)(op: (B, A) => B): B
  def foldRight[B](z: B)(op: (A, B) => B): B
  def foreach(f: A => Unit): Unit
  def iterator: Iterator[A]
```

```
  def drop(n: Int): List[A]
  def map[B](f: A => B): List[B]
  def flatMap[B](f: A => {*} IterableOnce[B]): List[B]
  def ++[B >: A](xs: {*} IterableOnce[B]): List[B]
```

Notably, these methods have almost exactly the same signatures as their versions in the standard Scala collections library. The only differences concern the arguments to `flatMap` and `++` which now admit an `IterableOnce` argument with an arbitrary capture set. Of course, we could have left out the `{*}` but this would have needlessly restricted the argument to non-capturing collections.

Contrast this with some of the same methods for iterators:

```
extension [A](it: Iterator[A])
  def apply(n: Int): A
  def foldLeft[B](z: B)(op: (B, A) => B): B
  def foldRight[B](z: B)(op: (A, B) => B): B
  def foreach(f: A => Unit): Unit

  def drop(n: Int): {it} Iterator[A]
  def map[B](f: A => B): {it, f} Iterator[B]
  def flatMap[B](f: A => {*} IterableOnce[B]): {it, f} Iterator[B]
  def ++[B >: A](xs: {*} IterableOnce[B]): {it, xs} Iterator[B]
```

Here, methods `apply`, `foldLeft`, `foldRight`, `foreach` again have the same signatures as in the current Scala standard library. But the remaining four operations need additional capture annotations. Method `drop` on iterators returns the given iterator `it` after skipping `n` elements. Consequently, its result has `{it}` as capture set. Methods `map` and `flatMap` lazily map elements of the current iterator as the result is traversed. Consequently they retain both `it` and `f` in their result capture set. Method `++` concatenates two iterators and therefore retains both of them in its result capture set.

The examples attest to the practicality of capture checking. Method signatures are generally concise. Higher-order methods over strict collections by and large keep the same types as before. Capture annotations are only needed for capabilities that are retained in closures and are executed on demand later. Where such annotations are needed they match the developer's intuitive understanding of reference patterns and signal information that looks relevant in this context.

6 RELATED WORK

The results presented in this paper did not emerge in a vacuum and many of the underlying ideas appeared individually elsewhere in similar or different form. We follow the structure of the informal presentation in Section 2 and organize the discussion of related work according to the key ideas behind $CC_{<:\Box}$.

Effects as Capabilities. Establishing effect safety by moderating access to effects via term-level capabilities is not a new idea [Marino and Millstein 2009]. It has been proposed as a strategy to retrofit existing languages with means to reason about effect safety [Choudhury and Krishnaswami 2020; Liu 2016; Osvald et al. 2016]. Recently, it also has been applied as the core principle behind a new programming language featuring effect handlers [Brachthäuser et al. 2020a]. Similar to the above prior work, we propose to use term-level capabilities to restrict access to effect operations and other scoped resources with a limited lifetime. Representing effects as capabilities results in a good economy of concepts: existing language features, like term-level binders can be reused, programmers are not confronted with a completely new concept of effects or regions.

Making Capture Explicit. Having a term-level representation of scoped capabilities introduces the challenge to restrict use of such capabilities to the scope in which they are still live. To address this issue, effect systems have been introduced [Biernacki et al. 2020; Brachthäuser et al. 2020b; Zhang and Myers 2019] but those can result in overly verbose and difficult to understand types [Brachthäuser et al. 2020a]. A third approach, which we follow in this paper, is to make capture explicit in the type of functions.

Hannan [1998] proposes a type-based escape analysis with the goal to facilitate stack allocation. The analysis tracks variable reference using a type-and-effect system and annotates every function type with the set of free variables it captures. The authors leave the treatment of effect polymorphism to future work. In a similar spirit, Scherer and Hoffmann [2013] present Open Closure Types to facilitate reasoning about data flow properties such as non-interference. They present an extension of the simply typed lambda calculus that enhances function types $[\Gamma_0](\tau) \to \tau$ with the lexical environment Γ_0 that was originally used to type the closure.

Brachthäuser et al. [2022] show System C, which mediates between first- and second-class values with boxes. In their system, scoped capabilities are second-class values. Normally, second-class values cannot be returned from any scope, but in System C they can be boxed and returned from *some* scopes. The type of a boxed second-class value tracks which scoped capabilities it has captured and accordingly, from which scopes it cannot be returned. System C tracks second-class values with a coeffect-like environment and uses an effect-like discipline for tracking captured capabilities, which can in specific cases be more precise than cv. In comparison, $\mathrm{CC}_{<:\Box}$ does not depend on a notion of second-class values and deeply integrates capture sets with subtyping.

Recently, Bao et al. [2021] have proposed to qualify types with *reachability sets*. Their *reachability types* allow reasoning about non-interference, scoping and uniqueness by tracking for each reference what other references it may alias or (indirectly) point to. Their system formalizes subtyping but not universal polymorphism. However, it relates reachability sets along a different dimension than $\mathrm{CC}_{<:\Box}$. Whereas in $\mathrm{CC}_{<:\Box}$ a subtyping relationship is established between a capability c and the capabilities in the type of c, reachability types assume a subtyping relationship between a variable x and the variable owning the scope where x is defined. Reachability types track detailed points-to and aliasing information in a setting with mutable variables, while $\mathrm{CC}_{<:\Box}$ is a more foundational calculus for tracking references and capabilities that can be used as a guide for an implementation in a complete programming language. It would be interesting to explore how reachability types can be expressed in $\mathrm{CC}_{<:\Box}$.

Capture Polymorphism. Combining effect tracking with higher-order functions immediately gives rise to effect polymorphism, which has been a long-studied problem.

Similar to the usual (parametric) type polymorphism, the seminal work by Lucassen and Gifford [1988] on type and effect systems featured (parametric) *effect polymorphism* by adding language constructs for explicit region abstraction and application. Similarly, work on region based memory management [Tofte and Talpin 1997] supports *region polymorphism* by explicit region abstraction and application. Recently, languages with support for algebraic effects and handlers, such as Koka [Leijen 2017] and Frank [Lindley et al. 2017], feature explicit, parametric effect polymorphism.

It has been observed multiple times, for instance by Osvald et al. [2016] and Brachthäuser et al. [2020a], that parametric effect polymorphism can become verbose and results in complicated types and confusing error messages. Languages sometimes attempt to *hide* the complexity – they "simplify the types more and leave out 'obvious' polymorphism" [Leijen 2017]. However, this solution is not satisfying since the full types resurface in error messages. In contrast, we support polymorphism by reusing existing term-level binders and support simplifying types by means of subtyping and subcapturing.

Rytz et al. [2012] present a type-and-effect system in which higher-order functions like `map` can be assigned simple signatures that do not mention effect variables. As in $CC_{<:\Box}$, it is not necessary to modify the signatures of higher-order functions which only call their argument. However, in the "argument-relative" system of Rytz et al., it is impossible to reference an effect of a particular argument. This limits the overall expressivity in their system, compared to $CC_{<:\Box}$ – for instance, it is not possible to type function composition, or in general a function that returns a value whose effect is relative to its argument. Their system also does not allow user-defined effects, while $CC_{<:\Box}$ allows tracking any variable by annotating it with an appropriate capture set.

The problem of how to prevent capabilities from escaping in closures is also addressed by *second-class values* that can only be passed as arguments but not be returned in results or stored in mutable fields. Siek et al. [2012] enforce second-class function arguments using a classical polymorphic effect discipline whereas Osvald et al. [2016] and Brachthäuser et al. [2020a] present a specialized type discipline for this task. Second-class values cannot be returned or closed-over by first-class functions. On the other hand, second-class functions can freely close over capabilities, since they are second-class themselves. This gives rise to a convenient and light-weight form of *contextual* effect polymorphism [Brachthäuser et al. 2020a]. While this approach allows for effect polymorphism with a simple type system, it is also restrictive because it also forbids local returns and retentions of capabilities; a problem solved by adding boxing and unboxing [Brachthäuser et al. 2022].

Foundations of Boxing. Contextual modal type theory (CMTT) [Nanevski et al. 2008] builds on intuitionistic modal logic. In intuitionistic modal logic, the graded propositional constructor $[\Psi]\ A$ (pronounced *box*) witnesses that A can be proven only using true propositions in Ψ. Judgements in CMTT have two contexts: Γ, roughly corresponding to $CC_{<:\Box}$ bindings with $\{*\}$ as their capture set, and a modal context Δ roughly corresponding to bindings with concrete capture sets. Bindings in the modal context are necessarily boxed and annotated with a modality $x :: A[\Psi] \in \Delta$. Just like our definition of captured variables in $CC_{<:\Box}$, the definition of free variables by Nanevski et al. [2008] assigns the empty set to a boxed term (that is, $fv(\mathsf{box}(\Psi.M)) = \{\}$). Similar to our unboxing construct, using a variable bound in the modal context requires that the current context satisfies the modality Ψ, mediated by a substitution σ. Different to CMTT, $CC_{<:\Box}$ does not introduce a separate modal context. It also does not annotate modalities on binders, instead these are kept in the types. Also different to CMTT, in $CC_{<:\Box}$ unboxing is annotated with a capture set and not a substitution.

Comonadic type systems were introduced to support reasoning about *purity* in existing, impure languages [Choudhury and Krishnaswami 2020]. Very similar to the box modality of CMTT, a type constructor 'Safe' witnesses the fact that its values are constructed without using any impure capabilities. The type system presented by Choudhury and Krishnaswami [2020] only supports a binary distinction between *pure* values and *impure* values, however, the authors comment that it might be possible to generalize their system to graded modalities.

In the present paper, we use boxing as a practical tool, necessary to obtain concise types when combining capture tracking with parametric type polymorphism.

Coeffect Systems. *Coeffect systems* also attach additional information to bindings in the environment, leading to a typing judgement of the form $\Gamma @ C \vdash e : \tau$. Such systems can be seen as similar in spirit to $CC_{<:\Box}$, where additional information is available about each variable in the environment through the capture set of its type. Petricek et al. [2014] show a general coeffect framework that can be instantiated to track various concepts such as bounded reuse of variables, implicit parameters and data access. This framework is based on simply typed lambda calculus and its function types are always coeffect-monomorphic. In contrast, $CC_{<:\Box}$ is based on System $F_{<:}$ (thus supporting type polymorphism and subtyping) and supports capture-polymorphic functions.

Object Capabilities. The (object-)capability model of programming [Boyland et al. 2001; Crary et al. 1999; Miller 2006], controls security critical operations by requiring access to a capability. Such a capability can be seen as the constructive proof that the holder is entitled to perform the critical operation. Reasoning about which operations a module can perform is reduced to reasoning about which references to capabilities a module holds.

The Wyvern language [Melicher et al. 2017] implements this model by distinguishing between stateful *resource modules* and *pure modules*. Access to resource modules is restricted and only possible through capabilities. Determining the authority granted by a module amounts to manually inspecting its type signature and all of the type signatures of its transitive imports. To support this analysis, Melicher [2020] extends the language with a fine-grained effect system that tracks access of capabilities in the type of methods.

Figueroa et al. [2016] show an intricately engineered encoding of object capabilities in Haskell. A monad transformer's private operations can only be called from modules with the appropriate authority. The capabilities may be part of a hierarchy, e.g. the ReadWrite capability may subsume the Read and Write capabilities. Capabilities may be shared between modules through encoded friend declarations, and one may determine a module's authority like in Wyvern.

In $CC_{<:\Box}$, one can statically reason about authority and capabilities simply by inspecting capture sets of types. Additionally, subcapturing naturally allows defining capability hierarchies. If we model modules via function abstraction, the function's capture set directly reflects its authority. Importantly, $CC_{<:\Box}$ does not include an effect system and thus tracks mention rather than use.

7 CONCLUSION

We introduced a new type system $CC_{<:\Box}$ to track captured references of values. Tracked references are restricted to capabilities, where capabilities are references bootstrapped from other capabilities, starting with the universal capability. Implementing this simple principle then naturally suggests a chain of design decisions:

(1) Because capabilities are variables, every function must have its type annotated with its free capability variables.
(2) To manage the scoping of those free variables, function types must be dependently-typed.
(3) To prevent non-variable terms from occurring in types, the programming language is formulated in monadic normal form.
(4) Because of type dependency, the let-bindings of MNF have to satisfy the avoidance property, to prevent out-of-scope variables from occurring in types.
(5) To make avoidance possible, the language needs a rich notion of subtyping on the capture sets.
(6) Because the capture sets represent object capabilities, the subcapture relation cannot just be the subset relation on sets of variables – it also has to take into account the types of the variables, since the variables may be bound to values which themselves capture capabilities.
(7) To keep the size of the capture sets from ballooning out of control, the paper introduces a box connective with box and unbox rules to control when free variables are counted as visible.

We showed that the resulting system can be used as the basis for lightweight polymorphic effect checking, without the need for effect quantifiers. We also identified three key principles that keep notational overhead for capture tracking low:

– Variables are tracked only if their types have non-empty capture sets. In practice the majority of variables are untracked and thus do not need to be mentioned at all.
– Subcapturing, subtyping and subsumption mean that more detailed capture sets can be subsumed by coarser ones.

– Boxed types stop propagation of capture information in enclosing types which avoids repetition in capture annotations to a large degree.

Our experience so far indicates that the presented calculus is simple and expressive enough to be used as a basis for more advanced effect and resource checking systems and their practical implementations.

ACKNOWLEDGMENTS

We thank Jonathan Aldrich, Vikraman Choudhury and Neal Krishnawami for their input in discussions about this research. We thank the anonymous reviewers of previous versions of this paper for their comments and suggestions. In particular, we paraphrased with permission one reviewer's summary of our design decisions in the conclusion. This research was partially funded by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World*. Springer, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–32. https://doi.org/10.1145/3485516

Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science* 6, 6 (Dec. 1996), 579–612. https://doi.org/10.1017/S0960129500070109

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, New York, NY, USA.

Corrado Böhm and Alessandro Berarducci. 1985. Automatic Synthesis of Typed λ-Programs on Term Algebras. *Theoretical Computer Science* 39 (1985), 135–154. https://doi.org/10.1016/0304-3975(85)90135-5

John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing. In *ECOOP 2001 — Object-Oriented Programming*, Jørgen Lindskov Knudsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–27.

Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, Capabilities, and Boxes: From Scope-based Reasoning to Type-based Reasoning and Back. https://se.informatik.uni-tuebingen.de/publications/brachthaeuser22effects/

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020a. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov. 2020). https://doi.org/10.1145/3428194

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020b. Effekt: Capability-Passing Style for Type- and Effect-safe, Extensible Effect Handlers in Scala. *Journal of Functional Programming* (2020). https://doi.org/10.1017/S0956796820000027

Vikraman Choudhury and Neel Krishnaswami. 2020. Recovering Purity with Comonads and Capabilities. *Proc. ACM Program. Lang.* 4, ICFP, Article 111 (Aug. 2020), 28 pages. https://doi.org/10.1145/3408993

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

William R. Cook. 2009. On Understanding Data Abstraction, Revisited. ACM, New York, NY, USA, 557–572.

Karl Crary, David Walker, and Greg Morrisett. 1999. Typed Memory Management in a Calculus of Capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 262–275. https://doi.org/10.1145/292540.292564

Ismael Figueroa, Nicolas Tabareau, and Éric Tanter. 2016. Effect Capabilities for Haskell: Taming Effect Interference in Monadic Programming. *Science of Computer Programming* 119 (April 2016), 3–30. https://doi.org/10.1016/j.scico.2015.11.010

Colin S. Gordon. 2020. Designing with Static Capabilities and Effects: Use, Mention, and Invariants (Pearl). In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:25. https://doi.org/10.4230/LIPIcs.ECOOP.2020.10

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 282–293. https://doi.org/10.1145/512529.512563

John Hannan. 1998. A Type-based Escape Analysis for Functional Languages. *Journal of Functional Programming* 8, 3 (May 1998), 239–273.

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* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 458–471. https://doi.org/10.1145/174675.178053

John Launchbury and Amr Sabry. 1997. Monadic State: Axiomatization and Type Safety. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming* (Amsterdam, The Netherlands) *(ICFP '97)*. Association for Computing Machinery, New York, NY, USA, 227–238. https://doi.org/10.1145/258948.258970

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

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. https://doi.org/10.1145/3009837.3009897

Fengyun Liu. 2016. A Study of Capability-Based Effect Systems. Master's thesis. infoscience.epfl.ch/record/219173

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

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. https://doi.org/10.1145/1481861.1481868

Darya Melicher. 2020. *Controlling Module Authority Using Programming Language Design.* Ph.D. Dissertation. Carnegie Mellon University.

Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A capability-based module system for authority control. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. Dissertation. Johns Hopkins University.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *ACM Trans. Comput. Logic* 9, 3, Article 23 (June 2008), 49 pages. https://doi.org/10.1145/1352582.1352591

James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *ECOOP'98 — Object-Oriented Programming (Lecture Notes in Computer Science)*, Eric Jul (Ed.). Springer, Berlin, Heidelberg, 158–185. https://doi.org/10.1007/BFb0054091

Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2018. Simplicitly: Foundations and Applications of Implicit Function Types. *Proc. ACM Program. Lang.* 2, POPL, Article 42 (Dec. 2018), 29 pages. https://doi.org/10.1145/3158130

Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondřej Lhoták. 2021. Safer Exceptions for Scala. In *Scala Symposium, Chicago, USA*. https://dl.acm.org/doi/10.1145/3486610.3486893

Martin Odersky and Guillaume Martres. 2020. Extension Methods. Scala 3 Language Reference Page. https://dotty.epfl.ch/docs/reference/contextual/extension-methods.html

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

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-Dependent Computation. In *Proceedings of the International Conference on Functional Programming* (Gothenburg, Sweden). ACM, New York, NY, USA, 123–135. https://doi.org/10.1145/2628136.2628160

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

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, Vol. 7313)*, James Noble (Ed.). Springer, 258–282. https://doi.org/10.1007/978-3-642-31057-7_13

Amr Sabry and Matthias Felleisen. 1993. Reasoning about Programs in Continuation-Passing Style. In *LISP AND SYMBOLIC COMPUTATION*. 288–298.

Scala. 2022a. Scala 3: Capture Checking. https://dotty.epfl.ch/docs/reference/experimental/cc.html

Scala. 2022b. The Scala 3 compiler, also known as Dotty. https://dotty.epfl.ch

Gabriel Scherer and Jan Hoffmann. 2013. Tracking Data-Flow with Open Closure Types. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 710–726. https://doi.org/10.1007/978-3-319-30936-1_14

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

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

Philip Wadler. 1990. Linear Types can Change the World!. In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, Manfred Broy and Cliff B. Jones (Eds.). North-Holland, 561.

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3, POPL, Article 5 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290318

A PROOFS

A.1 Proof devices

Throughout the proof, we follow the Barendregt convention and only consider environments where all variables are unique, i.e. if an environment is of the form $\Gamma, x : T$ we have $x \notin \mathrm{dom}(\Gamma)$.

We define environment well-formedness $\vdash\ \Gamma$ **wf** in the natural way – empty environment is well-formed, and if both $\vdash\ \Gamma$ **wf** and $\Gamma\ \vdash\ T$ **wf**, then also $\Gamma, x : T$ and $\Gamma, X <: T$.

To prove Preservation (Theorem A.43), we relate the typing derivation of a term of the form $\sigma[\ t\ ]$ to the typing derivation for the *plug* term t inside the store σ. We do so with the following definition:

Matching environment $\boxed{\Gamma\ \vdash\ \sigma \sim \Delta}$

$$\frac{\Gamma, x : T\ \vdash\ \sigma \sim \Delta \quad \Gamma\ \vdash\ v : T \quad x \notin \mathrm{fv}(T)}{\Gamma\ \vdash\ \mathbf{let}\, x = v\, \mathbf{in}\, \sigma \sim x : T, \Delta} \qquad \Gamma\ \vdash\ [\,] \sim \cdot$$

Definition A.1 (Evaluation context typing ($\Gamma \vdash e : U \Rightarrow T$)). We say that e can be typed as $U \Rightarrow T$ in Γ iff for all t such that $\Gamma \vdash t : U$, we have $\Gamma \vdash e[\ t\ ] : T$.

Fact A.2. *If $\sigma[\ t\ ]$ is a well-typed term in Γ, then there exists a Δ matching σ (i.e. such that $\Gamma\ \vdash\ \sigma \sim \Delta$), finding it is decidable, and Γ, Δ is well-formed.*

Fact A.3. *The analogous holds for $e[\ t\ ]$.*

A.2 Properties of Evaluation Contexts and Stores

In the proof, we use the following metavariables: C, D for capture sets, R, S for shape types, P, Q, T, U for types.

We also denote the capture set fragment of a type as $\mathrm{cv}(T)$, defined as $\mathrm{cv}(C\ R) = C$.

In all our statements, we implicitly assume that all environments are well-formed.

Lemma A.4 (Inversion of typing under an evaluation context). *If $\Gamma\ \vdash\ e[\ s\ ] : T$, then s is well-typed in Γ.*

Proof. Proceed by induction on e. If e is empty then the result directly holds, so consider when $s = \mathbf{let}\, x = e'[\ s'\ ]\, \mathbf{in}\, s''$. By inverting the typing derivation we know that $\Gamma\ \vdash\ e'[\ s'\ ] : T'$, and hence by induction we conclude that $\Gamma\ \vdash\ s : U$ for some type U. □

Lemma A.5 (Evaluation context typing inversion).
$\Gamma \vdash e[\ s\ ] : T$ *implies that for some* U *we have* $\Gamma \vdash e : U \Rightarrow T$ *and* $\Gamma \vdash s : U$.

Proof. By induction on the structure of e. If $e = [\ ]$, then $\Gamma \vdash s : T$ and clearly $\Gamma \vdash [\ ] : T \Rightarrow T$. Otherwise $e = \mathbf{let}\, x = e'\, \mathbf{in}\, t$. Proceed by induction on the typing derivation of $e[\ s\ ]$. We can only assume that $\Gamma \vdash e[\ s\ ] : T'$ for some T' s.t. $\Gamma \vdash T' <: T$.

Case (let). Then $\Gamma \vdash e'[\ s\ ] : U'$ and $\Gamma, x : U' \vdash t : T'$ for some U'. By the outer IH, for some U we then have $\Gamma \vdash e' : U \Rightarrow U'$ and $\Gamma \vdash s : U$. The former unfolds to $\forall s'.\, \Gamma \vdash s' : U \implies \Gamma \vdash e'[\ s'\ ] : U'$. We now want to show that $\forall s'.\, \Gamma \vdash s' : U \implies \Gamma \vdash e[\ s'\ ] : T'$. We already have $\Gamma \vdash e'[\ s'\ ] : U'$ and $\Gamma, x : U' \vdash t : T'$, so we can conclude by (let).

Case (sub). Then $\Gamma \vdash e[\ s\ ] : T''$ and $\Gamma \vdash T'' <: T'$. We can conclude by the inner IH and (trans).

□

Lemma A.6 (Evaluation context reification).
If both $\Gamma \vdash e : U \Rightarrow T$ *and* $\Gamma \vdash s : U$, *then* $\Gamma \vdash e[\ s\ ] : T$.

Proof. Immediate from the definition of $\Gamma \vdash e : U \Rightarrow T$. □

Lemma A.7 (Store context reification). *If* $\Gamma \vdash \sigma \sim \Delta$ *and* $\Gamma, \Delta \vdash t : T$ *then* $\Gamma \vdash \sigma[\ t\ ] : T$.

Proof. By induction on σ.

Case $\sigma = [\ ]$. Immediate.

Case $\sigma = \sigma'[\ \mathbf{let}\, x = v\, \mathbf{in}[\ ]\ ]$. Then $\Delta = \Delta', x : U$ for some U. Since $x \notin \mathrm{fv}(T)$ as $\Gamma \vdash T\ \mathbf{wf}$, by (let), we have that $\Gamma, \Delta' \vdash \mathbf{let}\, x = v\, \mathbf{in}\, t$ and hence by the induction hypothesis for some U we have that $\Gamma, x : U \vdash \sigma'[t] : T$. The result follows directly.

□

The above lemma immediately gives us:

Corollary A.8 (Replacement of term under a store context). *If* $\Gamma \vdash \sigma[\ t\ ] : T$ *and* $\Gamma \vdash \sigma \sim \Delta$ *and* $\Gamma, \Delta \vdash t : T$, *then for all* t' *such that* $\Gamma, \Delta \vdash t' : T$ *we have* $\Gamma \vdash \sigma[\ t'\ ] : T$.

A.3 Properties of Subcapturing

Lemma A.9 (Top capture set). *Let* $\Gamma \vdash C$ ***wf***. *Then* $\Gamma \vdash C <: \{*\}$.

Proof. By induction on Γ. If Γ is empty, then C is either empty or $* \in C$, so we can conclude by (sc-set) or (sc-elem) correspondingly. Otherwise, $\Gamma = \Gamma', x : D\,S$ and since Γ is well-formed, $\Gamma' \vdash D\ \mathbf{wf}$. By (sc-set), we can conclude if for all $y \in C$ we have $\Gamma \vdash \{y\} <: \{*\}$. If $y = x$, by IH we derive $\Gamma' \vdash D <: \{*\}$, which we then weaken to Γ and conclude by (sc-var). If $y \neq x$, then $\Gamma' \vdash \{y\}\ \mathbf{wf}$, so by IH we derive $\Gamma' \vdash \{y\} <: \{*\}$ and conclude by weakening. □

Corollary A.10 (Effectively top capture set). *Let* $\Gamma \vdash C, D$ ***wf*** *such that* $* \in D$. *Then we can derive* $\Gamma \vdash C <: D$.

Proof. We can derive $\Gamma \vdash C <: \{*\}$ by Lemma A.9 and then we can conclude by Lemma A.13 and (sc-elem). □

Lemma A.11 (Universal capability subcapturing inversion). *Let* $\Gamma \vdash C <: D$. *If* $* \in C$, *then* $* \in D$.

Proof. By induction on subcapturing. Case (sc-elem) immediate, case (sc-set) by repeated IH, case (sc-var) contradictory. □

Lemma A.12 (Subcapturing distributivity). *Let* $\Gamma \vdash C <: D$. *Then for all* $x \in C$ *we have* $\Gamma \vdash \{x\} <: D$.

Proof. By inspection of the last subcapturing rule used to derive $C <: D$. All cases are immediate. If the last rule was (sc-set), we have our goal as premise. Otherwise, we have $C = \{x\}$ and the goal follows directly. □

Lemma A.13 (Subcapturing transitivity). *If* $\Gamma \vdash C_1 <: C_2$ *and* $\Gamma \vdash C_2 <: C_3$ *then* $\Gamma \vdash C_1 <: C_3$.

Proof. By induction on the first derivation.
Case (sc-elem). $C_1 = \{x\}$ and $x \in C_2$, so by Lemma A.12 $\Gamma \vdash \{x\} <: C_3$.
Case (sc-var). Then $C_1 = \{x\}$ and $x : C_4\ R \in \Gamma$ and $\Gamma \vdash C_4 <: C_2$. By IH $\Gamma \vdash C_4 <: C_3$ and we can conclude by (sc-var).
Case (sc-set). By repeated IH and (sc-set).

□

Lemma A.14 (Subcapturing reflexivity). *If* $\Gamma \vdash C$ **wf**, *then* $\Gamma \vdash C <: C$.

Proof. By (sc-set) and (sc-elem). □

Lemma A.15 (Subtyping implies subcapturing). *If* $\Gamma \vdash C_1\ R_1 <: C_2\ R_2$, *then* $\Gamma \vdash C_1 <: C_2$.

Proof. By induction on the subtyping derivation. If (capt), immediate. If (trans), by IH and subcapturing transitivity Lemma A.13. If (refl), then $C_1 = C_2$ and we can conclude by Lemma A.14. Otherwise, $C_1 = C_2 = \{\}$ and we can conclude by (sc-set). □

A.3.1 Subtyping inversion.

Fact A.16. *Both subtyping and subcapturing are transitive.*

Proof. Subtyping is intrisically transitive through (trans), while subcapturing admits transitivity as per Lemma A.13. □

Fact A.17. *Both subtyping and subcapturing are reflexive.*

Proof. Again, this is an intrinsic property of subtyping by (refl) and an admissible property of subcapturing per Lemma A.14. □

Lemma A.18 (Subtyping inversion: type variable). *If* $\Gamma \vdash U <: C\ X$, *then* U *is of the form* $C'\ X'$ *and we have* $\Gamma \vdash C' <: C$ *and* $\Gamma \vdash X' <: X$.

Proof. By induction on the subtyping derivation.
Case (tvar), (refl). Follow from reflexivity (A.17).
Case (capt). Then we have $U = C'\ S$ and $\Gamma \vdash C' <: C$ and $\Gamma \vdash S <: X$.
This relationship is equivalent to $\Gamma \vdash \{\}\ S <: \{\}\ X$, on which we invoke the IH.
By IH we have $\{\}\ S = \{\}\ Y$ and we can conclude with $U = C'\ Y$.
Case (trans). Then we have $\Gamma \vdash U <: U$ and $\Gamma \vdash U <: C\ X$. We proceed by using the IH twice and conclude by transitivity (A.16).
Other rules are impossible.

□

Lemma A.19 (Subtyping inversion: capturing type). *If* $\Gamma \vdash U <: C\ S$, *then* U *is of the form* $C'\ S'$ *such that* $\Gamma \vdash C' <: C$ *and* $\Gamma \vdash S' <: S$.

Proof. We take note of the fact that subtyping and subcapturing are both transitive (A.16) and reflexive (A.17). The result follows from straightforward induction on the subtyping derivation. □

Lemma A.20 (Subtyping inversion: function type). *If* $\Gamma \vdash U <: C\ \forall(x : T_1)\ T_2$*, then* U *either is of the form* $C'\ X$ *and we have* $\Gamma \vdash C' <: C$ *and* $\Gamma \vdash X <: \forall(x : T_1)\ T_2$*, or* U *is of the form* $C'\ \forall(x : U_1)\ U_2$ *and we have* $\Gamma \vdash C' <: C$ *and* $\Gamma \vdash T_1 <: U_1$ *and* $\Gamma, x : T_1 \vdash U_2 <: T_2$*.*

Proof. By induction on the subtyping derivation.

Case (tvar). Immediate.

Case (fun), (refl). Follow from reflexivity (A.17).

Case (capt). Then we have $\Gamma \vdash C' <: C$ and $\Gamma \vdash S <: \forall(x : T_1)\ T_2$.

This relationship is equivalent to $\Gamma \vdash \{\}\ S <: \{\}\ \forall(x : T_1)\ T_2$, on which we invoke the IH.

By IH $\{\}\ S$ might have two forms. If $\{\}\ S = \{\}\ X$, then we can conclude with $U = C'\ X$.

Otherwise we have $\{\}\ S = \{\}\ \forall(x : U_1)\ U_2$ and $\Gamma \vdash T_1 <: U_1$ and $\Gamma, x : T_1 \vdash U_2 <: T_2$. Then, $U = C'\ \forall(x : U_1)\ U_2$ lets us conclude.

Case (trans). Then we have $\Gamma \vdash U <: U'$ and $\Gamma \vdash U <: C\ \forall(x : T_1)\ T_2$. By IH U may have one of two forms. If $U = C'\ X$, we proceed with Lemma A.18 and conclude by transitivity (A.16).

Otherwise $U = C'\ \forall(x : U_1)\ U_2$ and we use the IH again on $\Gamma \vdash U' <: C'\ \forall(x : U_1)\ U_2$. If $U = C''\ X$, we again can conclude by (A.16). Otherwise if $U = C''\ \forall(x : U_1)\ U_2$, the IH only gives us $\Gamma, x : U_1 \vdash U_2 <: U_2$, which we need to narrow to $\Gamma, x : T_1$ before we can similarly conclude by transitivity (A.16).

Other rules are not possible.

□

Lemma A.21 (Subtyping inversion: type function type). *If* $\Gamma \vdash U <: C\ \forall[X <: T_1]\ T_2$*, then* U *either is of the form* $C'\ X$ *and we have* $\Gamma \vdash C' <: C$ *and* $\Gamma \vdash X <: \forall[X <: T_1]\ T_2$*, or* U *is of the form* $C'\ \forall[X <: U_1]\ U_2$ *and we have* $\Gamma \vdash C' <: C$ *and* $\Gamma \vdash T_1 <: U_1$ *and* $\Gamma, X <: T_1 \vdash U_2 <: T_2$*.*

Proof. Analogous to the proof of Lemma A.20. □

Lemma A.22 (Subtyping inversion: boxed type). *If* $\Gamma \vdash U <: C\ \square\ T$*, then* U *either is of the form* $C'\ X$ *and we have* $\Gamma \vdash C' <: C$ *and* $\Gamma \vdash X <: \square\ T$*, or* U *is of the form* $C'\ \square\ U'$ *and we have* $\Gamma \vdash C' <: C$ *and* $\Gamma \vdash U' <: T$*.*

Proof. Analogous to the proof of Lemma A.20. □

A.3.2 Permutation, weakening, narrowing.

Lemma A.23 (Permutation). *Permutating the bindings in the environment up to preserving environment well-formedness also preserves type well-formedness, subcapturing, subtyping and typing.*

Let Γ *and* Δ *be the original and permutated context, respectively. Then:*

(1) If $\Gamma \vdash T$ **wf***, then* $\Delta \vdash T$ **wf***.*

(2) If $\Gamma \vdash C_1 <: C_2$*, then* $\Delta \vdash C_1 <: C_2$*.*

(3) If $\Gamma \vdash U <: T$*, then* $\Delta \vdash U <: T$*.*

(4) If $\Gamma \vdash t : T$*, then* $\Delta \vdash t : T$*.*

Proof. As usual, order of the bindings in the environment is not used in any rule. □

Lemma A.24 (Weakening). *Adding a binding to the environment such that the resulting environment is well-formed preserves type well-formedness, subcapturing, subtyping and typing.*

Let Γ *and* Δ *be the original and extended context, respectively. Then:*

(1) If $\Gamma \vdash T$ **wf**, *then* $\Delta \vdash T$ **wf**.
(2) If $\Gamma \vdash C_1 <: C_2$, *then* $\Delta \vdash C_1 <: C_2$.
(3) If $\Gamma \vdash U <: T$, *then* $\Delta \vdash U <: T$.
(4) If $\Gamma \vdash t : T$, *then* $\Delta \vdash t : T$.

Proof. As usual, the rules only check if a variable is bound in the environment and all versions of the lemma are provable by straightforward induction. For rules which extend the environment, such as (abs), we need permutation. All cases are analogous, so we will illustrate only one.

Case (abs). WLOG we assume that $\Delta = \Gamma, x : T$. We know that $\Gamma \vdash \lambda(y : U)t' : \forall(y : U)\ U$. and from the premise of (abs) we also know that $\Gamma, y : U \vdash t' : U$.

By IH, we have $\Gamma, y : U, x : T \vdash t' : U$. $\Gamma, x : T, y : U$ is still a well-formed environment (as T cannot mention y) and by permutation we have $\Gamma, x : T, y : U \vdash t' : U$. Then by (abs) we have $\Gamma, x : T \vdash \lambda(y : U)t' : \forall(y : U)\ U$, which concludes.

□

Lemma A.25 (Type binding narrowing).

(1) If $\Gamma \vdash S' <: S$ *and* $\Gamma, X <: S, \Delta \vdash T$ **wf**, *then* $\Gamma, X <: S', \Delta \vdash T$ **wf**.
(2) If $\Gamma \vdash S' <: S$ *and* $\Gamma, X <: S, \Delta \vdash C_1 <: C_2$, *then* $\Gamma, X <: S', \Delta \vdash C_1 <: C_2$.
(3) If $\Gamma \vdash S' <: S$ *and* $\Gamma, X <: S, \Delta \vdash T_1 <: T_2$, *then* $\Gamma, X <: S', \Delta \vdash T_1 <: T_2$.
(4) If $\Gamma \vdash S' <: S$ *and* $\Gamma, X <: S, \Delta \vdash t : T$, *then* $\Gamma, X <: S', \Delta \vdash t : T$.

Proof. By straightforward induction on the derivations. Only subtyping considers types to which type variables are bound, and the only rule to do so is (tvar), which we prove below. All other cases follow from IH or other narrowing lemmas.

Case (tvar). We need to prove $\Gamma, X <: S', \Delta \vdash X <: S$, which follows from weakening the lemma premise and using (trans) together with (tvar).

□

Lemma A.26 (Term binding narrowing).

(1) If $\Gamma \vdash U' <: U$ *and* $\Gamma, x : U, \Delta \vdash T$ **wf**, *then* $\Gamma, x : U', \Delta \vdash T$ **wf**.
(2) If $\Gamma \vdash U' <: U$ *and* $\Gamma, x : U, \Delta \vdash C_1 <: C_2$, *then* $\Gamma, x : U', \Delta \vdash C_1 <: C_2$.
(3) If $\Gamma \vdash U' <: U$ *and* $\Gamma, x : U, \Delta \vdash T_1 <: T_2$, *then* $\Gamma, x : U', \Delta \vdash T_1 <: T_2$.
(4) If $\Gamma \vdash U' <: U$ *and* $\Gamma, x : U, \Delta \vdash t : T$, *then* $\Gamma, x : U', \Delta \vdash t : T$.

Proof. By straightforward induction on the derivations. Only subcapturing and typing consider types to which term variables are bound. Only (sc-var) and (var) do so, which we prove below. All other cases follow from IH or other narrowing lemmas.

Case (var). We know that $U = C\ R$ and $\Gamma, x : C\ R, \Delta \vdash x : \{x\}\ R$. As $\Gamma \vdash U' <: U$, from Lemma A.19 we know that $U' = C'\ R'$ and that $\Gamma \vdash R' <: R$. We need to prove that $\Gamma, x : C'\ R', \Delta \vdash x : \{x\}\ R$. We can do so through (var), (sub), (capt), (sc-elem) and weakening $\Gamma \vdash R' <: R$.

Case (sc-var). Then we know that $C_1 = \{y\}$ and that $y : T \in \Gamma, x : U, \Delta$ and that $\Gamma, x : U, \Delta \vdash \mathrm{cv}(T) <: C_2$.

If $y \neq x$, we can conclude by IH and (sc-var).

Otherwise, we have $T = U$. From Lemma A.19 we know that $\Gamma \vdash \mathrm{cv}(U') <: \mathrm{cv}(U)$, and from IH we know that $\Gamma, x : U', \Delta \vdash \mathrm{cv}(U) <: C_2$. By (sc-var) to conclude it is enough to have $\Gamma, x : U', \Delta \vdash \mathrm{cv}(U') <: C_2$, which we do have by connecting two previous conclusions by weakening and Lemma A.13.

□

A.4 Substitution

A.4.1 Term Substitution. We will make use of the following fact:

Fact A.27. *If* $x : T \in \Gamma$ *and* $\vdash \Gamma$ ***wf****, then* $\Gamma = \Delta_1, x : T, \Delta_2$ *and* $\Delta_1 \vdash T$ ***wf*** *and so* $x \notin \mathrm{fv}(T)$.

Lemma A.28 (Term substitution preserves subcapturing). *If* $\Gamma, x : P, \Delta \vdash C_1 <: C_2$ *and* $\Gamma \vdash D <: cv(P)$*, then* $\Gamma, [x := D]\Delta \vdash [x := D]C_1 <: [x := D]C_2$.

Proof. Define $\theta \triangleq [x := D]$. By induction on the subcapturing derivation.

Case (SC-ELEM). Then $C_1 = \{y\}$ and $y \in C_2$. Inspect if $y = x$. If no, then our goal is $\Gamma, \theta\Delta \vdash \{y\} <: \theta C_2$. In this case, $y \in \theta C_2$, which lets us conclude by (SC-ELEM). Otherwise, we have $\theta C_2 = (C_2 \setminus \{x\}) \cup D$, as $x \in C_2$. Then our goal is $\Gamma, \theta\Delta \vdash D <: (C_2 \setminus \{x\}) \cup D$, which can be shown by (SC-SET) and (SC-ELEM).

Case (SC-VAR). Then $C_1 = \{y\}$ and $y : C_3\ S \in \Gamma, x : P, \Delta$ and $\Gamma, x : P, \Delta \vdash C_3 <: C_2$.

Inspect if $y = x$. If yes, then our goal is $\Gamma, \theta\Delta \vdash D <: \theta C_2$. By IH we know that $\Gamma, \theta\Delta \vdash \theta C_3 <: \theta C_2$. As $x = y$, we have $P = C_3\ S$ and therefore based on an initial premise of the lemma we have $\Gamma \vdash D <: C_3$. Then by weakening and IH, we know that $\Gamma, \theta\Delta \vdash \theta D <: \theta C_3$, which means we can conclude by Lemma A.13.

Otherwise, $x \neq y$, and our goal is $\Gamma, \theta\Delta \vdash C_1 <: \theta C_2$. We inspect where y is bound.

Case $y \in \mathrm{dom}(\Gamma)$. Then note that $y \notin C_3$ by Fact A.27. By IH we have $\Gamma, \theta\Delta \vdash \theta C_3 <: \theta C_2$. We can conclude by (SC-VAR) as $[x := D]C_3 = C_3$ and $y : C_3\ P \in \Gamma, \theta\Delta$.

Case $y \in \mathrm{dom}(\Delta)$. Then $y : \theta(C_3\ P) \in \Gamma, \theta\Delta$ and we can conclude by IH and (SC-VAR).

Case (SC-SET). Then $C_1 = \{y_1, \ldots, y_n\}$ and we inspect if $x \in C_1$.

If not, then for all $y \in C_1$ we have $\theta\{y\} = \{y\}$ and so we can conclude by repeated IH on our premises and (SC-SET).

If yes, then we know that: $\forall\, y \in C_1.\, \Gamma, x : P, \Delta \vdash \{y\} <: C_2$. We need to show that $\Gamma, \theta\Delta \vdash \theta C_1 <: \theta C_2$. By (SC-SET), it is enough to show that if $y' \in \theta C_1$, then $\Gamma, \theta\Delta \vdash \{y'\} <: \theta C_2$. For each such y', there exists $y \in C_1$ such that $y' \in \theta\{y\}$. For this y, from a premise of (SC-SET) we know that $\Gamma, x : P, \Delta \vdash \{y\} <: \theta C_2$ and so by IH we have $\Gamma, \theta\Delta \vdash \theta\{y\} <: \theta C_2$. Based on that, by Lemma A.13 we also have $\Gamma, \theta\Delta \vdash \{y'\} <: \theta C_2$. which is our goal.

□

Lemma A.29 (Term substitution preserves subtyping). *If* $\Gamma, x : P, \Delta \vdash U <: T$ *and* $\Gamma \vdash y : P$*, then* $\Gamma, [x := y]\Delta \vdash [x := y]U <: [x := y]T$.

Proof. Define $\theta \triangleq [x := y]$. Proceed by induction on the subtyping derivation.

Case (REFL), (TOP). By same rule.

Case (CAPT). By IH and Lemma A.31 and (CAPT).

Case (TRANS), (BOXED), (FUN), (TFUN). By IH and re-application of the same rule.

Case (TVAR). Then $U = Y$ and $T = S$ and $Y <: S \in \Gamma, x : U, \Delta$ and our goal is $\Gamma, \theta\Delta \vdash \theta Y <: \theta(S)$. Note that $x \neq Y$ and inspect where Y is bound. If $Y \in \mathrm{dom}(\Gamma)$, we have $Y <: S \in \Gamma, \theta\Delta$ and since $x \notin \mathrm{fv}(S)$ (Fact A.27), $\theta(S) = S$. Then, we can conclude by (TVAR). Otherwise if $Y \in \mathrm{dom}(\Delta)$, we have $Y <: \theta S \in \Gamma, \theta\Delta$ and again we can conclude by (TVAR).

□

Lemma A.30 (Term substitution preserves typing). *If* $\Gamma, x : P, \Delta \vdash t : T$ *and* $\Gamma \vdash x' : P$*, then* $\Gamma, [x := x']\Delta \vdash [x := x']t : [x := x']T$.

Proof. Define $\theta \triangleq [x := x']$. Proceed by induction on the typing derivation.

Case (VAR). Then $t = y$ and $y : C\ S \in \Gamma, x : P, \Delta$ and $T = \{y\}\ S$ and our goal is $\Gamma, \theta\Delta \vdash y : \theta(\{y\}\ S)$.

If $y = x$, then $P = C\,S$ and $\theta(\{x\}\,S) = \{x'\}\,S$. Our goal is $\Gamma, \theta\Delta \vdash x' : \{x'\}\,S$ and we can conclude by (VAR).

Otherwise, $y \neq x$ and we inspect where y is bound.

If $y \in \mathrm{dom}(\Gamma)$, then $x \notin \mathrm{fv}(C\,S)$ and so $\theta(\{z\}\,S) = \{z\}\,S$ and we can conclude by (VAR).

Otherwise, $y \in \mathrm{dom}(\Delta)$, so $y : \theta(C\,S) \in \Gamma, \theta\Delta$ and we can conclude by (VAR).

Case (SUB). By IH, Lemma A.29 and (SUB).

Case (ABS). Then $t = \lambda\,(y : Q).\ t', T = cv(t)\,\forall(y : Q)\ T'$ and $\Gamma, x : P, \Delta, y : Q \vdash t' : T'$.

By IH, we have that $\Gamma, \theta\Delta, y : \theta Q \vdash \theta t' : \theta T'$. We note that $\mathrm{cv}(\theta t) = \theta\,\mathrm{cv}(t)$, which lets us conclude by (ABS).

Case (TABS). Similar to previous rule.

Case (APP). Then $t = z_1\,z_2$ and $\Gamma, x : P, \Delta \vdash z_1 : C\,\forall(y : Q)\ T'$ and $\Gamma, x : P, \Delta \vdash z_1 : Q$ and $T = [y := z_2]T'$.

By IH we have $\Gamma, \theta\Delta \vdash \theta z_1 : \theta(C\,\forall(y : Q)\ T')$ and $\Gamma, \theta\Delta \vdash \theta z_2 : \theta Q$.

Then by (APP) we have $\Gamma, \theta\Delta \vdash \theta(z_1\,z_2) : [y := \theta z_2]\theta T'$.

As $y \neq x$ and $y \neq x'$, we have $[y := \theta z_2]\theta T' = \theta([y := z_2]T')$, which concludes.

Case (TAPP). Similar to previous rule.

Case (BOX). Then $t = \Box\,z$ and $\Gamma, x : P, \Delta \vdash z : C\,S$ and $T = \Box\,C\,S$.

By IH, we have $\Gamma, \theta\Delta \vdash \theta z : \theta C\,\theta S$. If $x \notin C$, we have $\theta C = C$ and $C \subseteq \mathrm{dom}(\Gamma, \theta\Delta)$ which lets us conclude by (BOX). Otherwise, $\theta C = (C \setminus \{x\}) \cup \{y\}$ As $\Gamma \vdash y : U$, $\theta C \subseteq \mathrm{dom}(\Gamma, \theta\Delta)$, which again lets us conclude by (BOX).

Case (UNBOX). Analogous to the previous rule. Note that we just swap the types in the premise and the conclusion.

Case (LET). Then $t = \mathbf{let}\,y = s\,\mathbf{in}\,t'$ and $\Gamma, x : P, \Delta \vdash s : Q$ and $\Gamma, x : P, \Delta, y : Q \vdash t' : T$. By the IH, we have $\Gamma, \theta\Delta \vdash \theta s : \theta Q$ and $\Gamma, \theta\Delta, y : \theta Q \vdash \theta t' : \theta T$.

Then by (LET) we also have $\Gamma, \theta\Delta \vdash \theta(\mathbf{let}\,y = s\,\mathbf{in}\,t') : \theta T$, which concludes. □

A.4.2 Type Substitution.

LEMMA A.31 (TYPE SUBSTITUTION PRESERVES SUBCAPTURING). *If* $\Gamma, X <: S, \Delta \vdash C <: D$ *and* $\Gamma \vdash R <: S$ *then* $\Gamma, [X := R]\Delta \vdash C <: D$.

PROOF. Define $\theta \triangleq [X := R]$. Proceed by induction on the subcapturing derivation.

Case (SC-SET), (SC-ELEM). By IH and same rule.

Case (SC-VAR). Then $C = \{y\}$, $y : C'\,S' \in \Gamma, X <: S, \Delta$, $y \neq X$. Inspect where y is bound. If $y \in \mathrm{dom}(\Gamma)$, we have $y : C'\,S' \in \Gamma, \theta\Delta$. Otherwise, by definition of substition we have $y : C'\,\theta S' \in \Gamma, \theta\Delta$. In both cases we can conclude by (SC-VAR), since y is still bound to a type whose capture set is C'.

□

LEMMA A.32 (TYPE SUBSTITUTION PRESERVES SUBTYPING). *If* $\Gamma, X <: S, \Delta \vdash U <: T$ *and* $\Gamma \vdash R <: S$, *then* $\Gamma, [X := R]\Delta \vdash [X := R]U <: [X := R]T$.

PROOF. Define $\theta \triangleq [X := R]$. Proceed by induction on the subtyping derivation.

Case (REFL), (TOP). By same rule.

Case (CAPT). By IH and Lemma A.31 and (CAPT).

Case (TRANS), (BOXED), (FUN), (TFUN). By IH and re-application of the same rule.

Case (TVAR). Then $U = Y$ and $T = S'$ and $Y <: S' \in \Gamma, X <: S, \Delta$ and our goal is $\Gamma, X <: S, \Delta \vdash \theta Y <: \theta S'$. If $Y = X$, by lemma premise and weakening. Otherwise, inspect where Y is bound. If $Y \in \mathrm{dom}(\Gamma)$, we have $Y <: S' \in \Gamma, \theta\Delta$ and since $X \notin \mathrm{fv}(S')$ (Fact A.27), $\theta S' = S'$. Then, we can conclude by (TVAR). Otherwise if $Y \in \mathrm{dom}(\Delta)$, we have $Y <: \theta S' \in \Gamma, \theta\Delta$ and we can conclude by (TVAR).

□

Lemma A.33 (Type substitution preserves typing). *If* $\Gamma, X <: S, \Delta \vdash t : T$ *and* $\Gamma \vdash R <: S$, *then* $\Gamma, [X := R]\Delta \vdash [X := R]t : [X := R]T$.

Proof. Define $\theta \triangleq [X := R]$. Proceed by induction on the typing derivation.

Case (var). Then $t = y$, $y : C\ S' \in \Gamma, X <: S, \Delta$, $y \neq X$, and our goal is $\Gamma, \theta\Delta \vdash y : \{y\}\ \theta S'$.

Inspect where y is bound. If $y \in \mathrm{dom}(\Gamma)$, then $y : C\ S' \in \Gamma, \theta\Delta$ and $X \notin \mathrm{fv}(S')$ (Fact A.27). Then, $\theta(C\ S') = C\ S'$ and we can conclude by (var). Otherwise, $y : C\ \theta S' \in \Gamma, \theta\Delta$ and we can directly conclude by (var).

Case (abs), (tabs). In both rules, observe that type substitution does not affect cv and conclude by IH and rule re-application.

Case (app). Then we have $t = x\ y$ and $\Gamma, X <: S, \Delta \vdash x : C\ \forall(z : U)\ T_0$ and $T = [z := y]T_0$.

We observe that $\theta[z := y]T_0 = [z := y]\theta T_0$ and $\theta t = t$ and conclude by IH and (app).

Case (tapp). Then we have $t = x\ [S']$ and $\Gamma, X <: S, \Delta \vdash x : C\ \forall[Z <: S']\ T_0$ and $T = [Z := S']T_0$.

We observe that $\theta[Z := S']T_0 = [Z := \theta S']\theta T_0$. By IH, $\Gamma, \theta\Delta \vdash x : C\ \forall[Z <: \theta S']\ T_0$, Then, we can conclude by (tapp).

Case (box). Then $t = \square\ y$ and $\Gamma, X <: S, \Delta \vdash y : C\ S'$ and $T = \square\ C\ S'$, and our goal is $\Gamma, \theta\Delta \vdash y : \square\ \theta(C\ S')$.

Inspect where y is bound. If $y \in \mathrm{dom}(\Gamma)$, then $y : C\ S' \in \Gamma, \theta\Delta$ and $X \notin \mathrm{fv}(S')$ (Fact A.27) . Then, $\theta(C'\ S') = C'\ S'$ and we can conclude by (box). Otherwise, $y : C\ \theta S' \in \Gamma, \theta\Delta$ and we can directly conclude by (box).

Case (unbox). Proceed analogously to the case for (box) – we just swap the types in the premise and in the consequence.

Case (sub). By IH and A.32.

Case (let). Then $t = \mathbf{let}\ y = s\ \mathbf{in}\ t'$ and $\Gamma, x : P, \Delta \vdash s : Q$ and $\Gamma, x : P, \Delta, y : Q \vdash t' : T$. By the IH, we have $\Gamma, \theta\Delta \vdash \theta s : \theta Q$ and $\Gamma, \theta\Delta, y : \theta Q \vdash \theta t' : \theta T$.

Then by (let) we also have $\Gamma, \theta\Delta \vdash \theta(\mathbf{let}\ y = s\ \mathbf{in}\ t') : \theta T$, which concludes. □

A.5 Main Theorems – Soundness

A.5.1 Preliminaries. As we state Preservation (Theorem A.43) in a non-empty environment, we need to show canonical forms lemmas in such an environment as well. To do so, we need to know that values cannot be typed with a type that is a type variable, which normally follows from the environment being empty. Instead, we show the following lemma:

Lemma A.34 (Value typing). *If* $\Gamma \vdash v : T$, *then* T *is not of the form* $C\ X$.

Proof. By induction on the typing derivation.

For rule (sub), we know that $\Gamma \vdash v : U$ and $\Gamma \vdash U <: T$. Assuming $T = C\ X$, we have a contradiction by Lemma A.18 and IH.

Rules (box), (abs), (tabs) are immediate, and other rules are not possible. □

Lemma A.35 (Canonical forms: term abstraction). *If* $\Gamma \vdash v : C\ \forall(x : U)\ T$, *then we have* $v = \lambda(x : U')t$ *and* $\Gamma \vdash U <: U'$ *and* $\Gamma, x : U \vdash t : T$.

Proof. By induction on the typing derivation.

For rule (sub), we observe that by Lemma A.20 and by Lemma A.34, the subtype is of the form $C'\ \forall(y : U'')\ T'$ and we have $\Gamma \vdash U <: U''$. By IH we know that $v = \lambda(x : U')t$ and $\Gamma \vdash U'' <: U'$ and $\Gamma, x : U'' \vdash t : T$. By (trans) we have $\Gamma \vdash U <: U'$ and by narrowing we have $\Gamma, x : U \vdash t : T$, which concludes.

Rule (ABS) is immediate, and other rules cannot occur. □

Lemma A.36 (Canonical forms: type abstraction). *If* $\Gamma \vdash v : C\ \forall[X <: U]\ T$, *then we have* $v = \lambda[X <: U']t$ *and* $\Gamma \vdash U <: U'$ *and* $\Gamma, X <: U \vdash t : T$.

Proof. Analogous to the proof of Lemma A.35. □

Lemma A.37 (Canonical forms: boxed term). *If* $\Gamma \vdash v : C\ \square\ T$, *then* $v = \square\ x$ *and* $\Gamma \vdash x : T$.

Proof. Analogous to the proof of Lemma A.35. □

Lemma A.38 (Variable typing inversion). *If* $\Gamma \vdash x : C\ S$, *then* $x : C'\ S' \in \Gamma$ *and* $\Gamma \vdash S' <: S$ *for some* C', S'.

Proof. By induction on the typing derivation.

Case (SUB). Then $\Gamma \vdash x : C''\ S''$ and $\Gamma \vdash C''\ S'' <: C\ S$. By the IH we have $\Gamma \vdash x : C'\ S'$ and $\Gamma \vdash S' <: S''$. Then by Lemma A.19 we have $\Gamma \vdash S'' <: S$, which lets us conclude by (TRANS).

Case (VAR). Then $\Gamma \vdash x : C'\ S$ and $C = \{x\}$. We can conclude with $S' = S$ by (REFL). □

Lemma A.39 (Variable lookup inversion). *If we have both* $\Gamma \vdash \sigma \sim \Delta$ *and* $x : C\ S \in \Gamma, \Delta$, *then* $\sigma(x) = v$ *implies that* $\Gamma, \Delta \vdash v : C\ S$.

Proof. By structural induction on σ. It is not possible for σ to be empty.

Otherwise, $\sigma = \sigma'[\ \textbf{let}\ y = v\ \textbf{in}[\ ]\ ]$ and for some U we have both $\Delta = \Delta', y : U$ and $\Gamma, \Delta' \vdash v : U$.

If $y \neq x$, we can proceed by IH as x can also be typed in Γ, Δ', after which we can conclude by weakening. Otherwise, $U = C\ S$ and we can conclude by weakening. □

Lemma A.40 (Term abstraction lookup inversion). *If* $\Gamma \vdash \sigma \sim \Delta$ *and* $\Gamma, \Delta \vdash x : C\ \forall(z : U)\ T$ *and* $\sigma(x) = \lambda(z : U')t$, *then* $\Gamma, \Delta \vdash U <: U'$ *and* $\Gamma, \Delta, z : U \vdash t : T$.

Proof. A corollary of Lemma A.39 and Lemma A.35. □

Lemma A.41 (Type abstraction lookup inversion). *If* $\Gamma \vdash \sigma \sim \Delta$ *and* $\Gamma, \Delta \vdash x : C\ \forall[Z <: U]\ T$ *and* $\sigma(x) = \lambda[Z <: U']t$, *then* $\Gamma, \Delta \vdash U <: U'$ *and* $\Gamma, \Delta, Z <: U \vdash t : T$.

Proof. A corollary of Lemma A.39 and Lemma A.36. □

Lemma A.42 (Box lookup inversion). *If* $\Gamma \vdash \sigma \sim \Delta$ *and* $\sigma(x) = \square\ y$ *and* $\Gamma, \Delta \vdash x : \square\ T$, *then* $\Gamma, \Delta \vdash y : T$.

Proof. A corollary of Lemma A.39 and Lemma A.37. □

A.5.2 Soundness. In this section, we show the classical soundness theorems.

Theorem A.43 (Preservation). *If we have* $\Gamma \vdash \sigma \sim \Delta$ *and* $\Gamma, \Delta \vdash t : T$, *then* $\sigma[\ t\ ] \longrightarrow \sigma[\ t'\ ]$ *implies that* $\Gamma, \Delta \vdash t' : T$.

Proof. We proceed by inspecting the rule used to reduce $\sigma[\ t\ ]$.

Case (APPLY). Then we have $t = e[\ x\ y\ ]$ and $\sigma(x) = \lambda(z : U)s$ and $t' = e[\ [z := y]s\ ]$.

By Lemma A.4, for some Q we have $\Gamma, \Delta \vdash e : Q \Rightarrow T$ and $\Gamma, \Delta \vdash x\ y : Q$. The typing derivation of $x\ y$ must start with an arbitrary number of (SUB) rules, followed by (APP). We proceed by induction on the number of (SUB) rules. In both base and inductive cases we can only assume that $\Gamma, \Delta \vdash x\ y : Q'$ for some Q' such that $\Gamma, \Delta \vdash Q' <: Q$.

In the inductive case, $\Gamma, \Delta \vdash x\ y : Q'$ is derived by (SUB), so we also have some Q'' such that $\Gamma, \Delta \vdash x\ y : Q''$ and $\Gamma, \Delta \vdash Q'' <: Q'$. We have $\Gamma, \Delta \vdash Q'' <: Q$ by (TRANS), so we can conclude by using the inductive hypothesis on $\Gamma, \Delta \vdash x\ y : Q''$.

In the base case, $\Gamma, \Delta \vdash x\, y : Q'$ is derived by (APP), so for some Q'' we have $\Gamma, \Delta \vdash x : \forall(z : U')\ Q''$ and $\Gamma, \Delta \vdash y : U'$ and $Q' = [z := y]Q''$. By Lemma A.40 and narrowing, we have $\Gamma, \Delta, z : U' \vdash s : Q''$. By Lemma A.30, we have $\Gamma, \Delta \vdash [z := y]s : [z := y]Q''$, and since $Q' = [z := y]Q''$, by (SUB) we have $\Gamma, \Delta \vdash [z := y]s : Q$.

To conclude that $t' = e[\ [z := y]s\ ]$ can be typed as T, we use Lemma A.6.

Case (TAPPLY), (OPEN). As above.

Case (RENAME). Then we have $t = e[\ \mathbf{let}\, x = y\, \mathbf{in}\, s\ ]$ and $t' = e[\ [x := y]s\ ]$.

Again, by Lemma A.4 for some Q we have $\Gamma, \Delta \vdash e : Q \Rightarrow T$ and $\Gamma, \Delta \vdash \mathbf{let}\, x = y\, \mathbf{in}\, s : Q$.

We again proceed by induction on number of (SUB) rules at the start of the typing derivation for $\mathbf{let}\, x = y\, \mathbf{in}\, s$, again only assuming that we can type the plug as some Q' such that $Q' <: Q$. The inductive case proceeds exactly as before.

In the base case, (LET) was used to derive that $\Gamma, \Delta \vdash \mathbf{let}\, x = y\, \mathbf{in}\, s : Q'$. The premises are $\Gamma, \Delta \vdash y : U$ and $\Gamma, \Delta, x : U \vdash s : Q'$ and $x \notin \mathrm{fv}(Q')$. By Lemma A.30, we have $\Gamma, \Delta \vdash [x := y]s : [x := y]Q'$. Because $x \notin \mathrm{fv}(Q')$, $[x := y]Q' = Q'$, which means that we can again conclude by (SUB) and Lemma A.6.

Case (LIFT). Then we have $t = e[\ \mathbf{let}\, x = v\, \mathbf{in}\, s\ ]$ and $t' = \mathbf{let}\, x = v\, \mathbf{in}\, e[\ s\ ]$.

Again, by Lemma A.4 for some Q we have $\Gamma, \Delta \vdash e : Q \Rightarrow T$ and $\Gamma, \Delta \vdash \mathbf{let}\, x = y\, \mathbf{in}\, s : Q$.

We again proceed by induction on number of (SUB) rules at the start of the typing derivation for $\mathbf{let}\, x = v\, \mathbf{in}\, s$, again only assuming that we can type the plug as some Q' such that $Q' <: Q$. The inductive case proceeds exactly as before.

In the base case, (LET) was used to derive that $\Gamma, \Delta \vdash \mathbf{let}\, x = v\, \mathbf{in}\, s : Q'$. The premises are $\Gamma, \Delta \vdash v : U$ and $\Gamma, \Delta, x : U \vdash s : Q'$ and $x \notin \mathrm{fv}(Q')$.

By weakening of typing, we also have $\Gamma, \Delta, x : U \vdash e : Q \Rightarrow T$. Then by (SUB) and Lemma A.6 we have $\Gamma, \Delta, x : U \vdash e[\ s\ ] : T$. Since $\Gamma, \Delta \vdash T\ \mathbf{wf}$, by Barendregt $x \notin \mathrm{fv}(T)$, so by (LET) we have $\Gamma, \Delta \vdash \mathbf{let}\, x = v\, \mathbf{in}\, e[\ s\ ] : T$, which concludes.

□

Definition A.44 (Canonical store-plug split). We say that a term of the form $\sigma[\ t\ ]$ is a *canonical split* (of the entire term into store context σ and the plug t) if t is not of the form $\mathbf{let}\, x = v\, \mathbf{in}\, t'$.

FACT A.45. *Every term has a unique canonical store-plug split, and finding it is decidable.*

LEMMA A.46 (EXTRACTION OF BOUND VALUE). *If* $\Gamma, \Delta \vdash x : T$ *and* $\vdash \sigma \sim \Delta$ *and* $x \in \mathrm{dom}(\Delta)$, *then* $\sigma(x) = v$.

PROOF. By structural induction on Δ. If Δ is empty, we have a contradiction. Otherwise, $\Delta = \Delta', z : T'$ and $\sigma = \sigma'[\ \mathbf{let}\, z = v\, \mathbf{in}[\ ]\ ]$ and $\Gamma, \Delta', z : T' \vdash v : T'$. Note that Δ is the environment matching σ and can only contain term bindings. If $z = x$, we can conclude immediately, and otherwise if $z \neq x$, we can conclude by IH. □

THEOREM A.47 (PROGRESS). *If* $\vdash \sigma[\ e[\ t\ ]\ ] : T$ *and* $\sigma[\ e[\ t\ ]\ ]$ *is a canonical store-plug split, then either* $e[\ t\ ] = a$, *or there exists* $\sigma[\ t'\ ]$ *such that* $\sigma[\ e[\ t\ ]\ ] \longrightarrow \sigma[\ t'\ ]$.

PROOF. Since $\sigma[\ e[\ t\ ]\ ]$ is well-typed in the empty environment, there clearly must be some Δ such that $\vdash \sigma \sim \Delta$ and $\Delta \vdash e[\ t\ ] : T$. By Lemma A.4, we have that $\Delta \vdash t : P$ for some P. We proceed by induction on the derivation of this derivation.

Case (VAR). Then $t = x$.

If e is non-empty, $e[\ x\ ] = e'[\ \mathbf{let}\, y = x\, \mathbf{in}\, t\ ]$ and we can step by (RENAME); otherwise, immediate.

Case (ABS), (TABS), (BOX). Then $t = v$.

If e is non-empty, $e[\ v\ ] = e'[\ \mathbf{let}\, x = v\, \mathbf{in}\, t\ ]$ and we can step by (LIFT); otherwise, immediate.

Case (APP). Then $t = x\, y$ and $\Delta \vdash x : C\ \forall(z : U)\ T_0$ and $\Delta \vdash y : U$.

By Lemmas A.46 and A.35, $\sigma(x) = \lambda(z : U')t'$, which means we can step by (APPLY).

Case (TAPP). Then $t = x\ [S]$ and $\Delta \vdash x : C\ \forall[Z <: S]\ T_0$.

By Lemmas A.46 and A.36, $\sigma(x) = \Lambda\ [z <: S']\ .\ t'$, which means we can step by (TAPPLY).

Case (UNBOX). Then $t = C \multimap x$ and $\Delta \vdash x : C \,\square\, C\, S$.

By Lemmas A.46 and A.37, $\sigma(x) = \square\ y$, which means we can step by (OPEN).

Case (LET). Then $t = \mathbf{let}\, x = s\, \mathbf{in}\, t'$ and we proceed by IH on s, with $e[\ \mathbf{let}\, x = [\ ]\, \mathbf{in}\, t'\ ]$ as the evaluation context.

Case (SUB). By IH.

□

A.5.3 Consequences.

LEMMA A.48 (CAPTURE PREDICTION FOR ANSWERS). *If* $\Gamma \vdash \sigma[\ a\ ] : C\ S$, *then* $\Gamma \vdash \sigma[\ a\ ] : \mathrm{cv}(\sigma[\ a\ ])\ S$ *and* $\Gamma \vdash \mathrm{cv}(\sigma[\ a\ ]) <: C$.

PROOF. By induction on the typing derivation.

Case (SUB). Then $\Gamma \vdash a : C'\ S'$ and $\Gamma \vdash C'\ S' <: C\ S$. By IH, $\Gamma \vdash \sigma[\ a\ ] : \mathrm{cv}(\sigma[\ a\ ])\ S'$ and $\Gamma \vdash \mathrm{cv}(\sigma[\ a\ ]) <: C'$. By Lemma A.19, we have that $\Gamma \vdash C' <: C$ and $\Gamma \vdash S' <: S$.

To conclude we need $\Gamma \vdash \sigma[\ a\ ] : \mathrm{cv}(\sigma[\ a\ ])\ S$ and $\Gamma \vdash \mathrm{cv}(\sigma[\ a\ ]) <: C$, which we respectively have by subsumption and Lemma A.13.

Case (VAR), (ABS), (TABS), (BOX). Then σ is empty and $C = \mathrm{cv}(a)$. One goal is immediate, other follows from Lemma A.14.

Case (LET). Then $\sigma = \mathbf{let}\, x = v\, \mathbf{in}\, \sigma'$ and $\Gamma, x : U \vdash \sigma'[\ a\ ] : C\ S$ and $x \notin C$.

By IH, $\Gamma, x : U \vdash \sigma'[\ a\ ] <: \mathrm{cv}(\sigma'[\ a\ ])\ S$ and $\Gamma, x : U \vdash \mathrm{cv}(\sigma'[\ a\ ]) <: C$.

By Lemma A.28, we have $\Gamma \vdash [x := \mathrm{cv}(v)](\mathrm{cv}(\sigma'[\ a\ ])) <: [x := \mathrm{cv}(v)]C$.

By definition, $[x := \mathrm{cv}(v)](\mathrm{cv}(\sigma'[\ a\ ])) = \mathrm{cv}(\mathbf{let}\, x = v\, \mathbf{in}\, \sigma'[\ a\ ])$, and we also already know that $x \notin C$.

This lets us conclude, as we have $\Gamma \vdash \mathrm{cv}(\mathbf{let}\, x = v\, \mathbf{in}\, \sigma'[\ a\ ]) <: C$.

Other rules cannot occur.

□

LEMMA A.49 (CAPTURE PREDICTION FOR TERMS). *Let* $\vdash \sigma \sim \Delta$ *and* $\Delta \vdash t : C\ S$. *Then* $e[\ t\ ] \longrightarrow^* e[\ \sigma'[\ a\ ]\ ]$ *implies that* $\Delta \vdash \mathrm{cv}(\sigma'[\ a\ ]) <: C$.

PROOF. By preservation, $\vdash \sigma'[\ a\ ] : C\ S$, which lets us conclude by Lemma A.48. □

A.6 Correctness of boxing

A.6.1 Relating cv and stores. We want to relate the cv of a term of the form $\sigma[\ t\ ]$ with $\mathrm{cv}(t)$ such that, for some definition of 'resolve', we have:

$$\mathrm{cv}(\sigma[\ t\ ]) = \mathrm{resolve}(\sigma, \mathrm{cv}(t))$$

Let us consider term of the form $\sigma[\ t\ ]$ and a store σ of the form $\mathbf{let}\, x = v\, \mathbf{in}\, \sigma'$. There are two rules that could be used to calculate $\mathrm{cv}(\mathbf{let}\, x = v\, \mathbf{in}\, \sigma')$:

$$\mathrm{cv}(\mathbf{let}\, x = v\, \mathbf{in}\, t) = \mathrm{cv}(t) \qquad \mathbf{if}\ x \notin \mathrm{cv}(t)$$
$$\mathrm{cv}(\mathbf{let}\, x = s\, \mathbf{in}\, t) = \mathrm{cv}(s) \cup \mathrm{cv}(t) \backslash x$$

Observe that since we know that x is bound to a value, we can reformulate these rules as:

$$\mathrm{cv}(\mathbf{let}\, x = v\, \mathbf{in}\, t) = [x := \mathrm{cv}(v)]\, \mathrm{cv}(t)$$

Which means that we should be able to define 'resolve' with a substitution. We will call this substitition a *store resolver*, and we define it as:

$$\mathrm{resolver}(\mathbf{let}\, x = v\, \mathbf{in}\, \sigma) = [x := \mathrm{cv}(v)] \circ \mathrm{resolver}(\sigma)$$
$$\mathrm{resolver}([\,]) = id$$

Importantly, note that we use *composition* of substitutions. We have:

$$\mathrm{resolver}(\mathbf{let}\, x = a\, \mathbf{in}\, \mathbf{let}\, y = x\, \mathbf{in}[\,]) \equiv [x := \{a\}, y := \{a\}]$$

With the above, we define resolve as:

$$\mathrm{resolve}(\sigma, C) = \mathrm{resolver}(\sigma)(C)$$

This definition satisfies our original desired equality with cv:

Fact A.50. *For all terms t of the form $\sigma[\, s\,]$, we have* $\mathrm{cv}(t) = \mathrm{resolve}(\sigma, \mathrm{cv}(s))$

A.6.2 Relating cv and evaluation contexts. We now relate cv to evaluation contexts e. First, note that by definition of cv we have:

Fact A.51. *For all terms t of the form $\mathbf{let}\, x = s\, \mathbf{in}\, t'$ such that s is not a value, we have* $\mathrm{cv}(t) = cv(s) \cup \mathrm{cv}(t') \setminus x$.

Accordingly, we extend cv to evaluation contexts ($\mathrm{cv}(e)$) as follows:

$$\mathrm{cv}(\mathbf{let}\, x = e\, \mathbf{in}\, t) = \mathrm{cv}(e) \cup \mathrm{cv}(t) \setminus x$$
$$\mathrm{cv}([\,]) = \{\}$$

We then have:

Fact A.52. *For all terms t of the form $e[\, s\,]$ such that s is not a value, we have* $\mathrm{cv}(t) = \mathrm{cv}(e) \cup \mathrm{cv}(s)$.

A.6.3 Relating cv to store and evaluation context simultaneously. Given our definition of 'resolve' and $\mathrm{cv}(e)$, we have:

Fact A.53. *Let $\sigma[\, e[\, t\,]\,]$ be a term such that t is not a value. Then:*

$$\mathrm{cv}(\sigma[\, e[\, t\,]\,]) = \mathrm{resolve}(\sigma, \mathrm{cv}(e) \cup \mathrm{cv}(t))$$

The proof proceeds by induction on σ and e, using Facts A.50 and A.52.

A.6.4 Correctness of cv.

Definition A.54 (Platform environment).
Γ is a platform environment if for all $x \in \mathrm{dom}(\Gamma)$ we have $x : \{*\}\, S \in \Gamma$ for some S.

Lemma A.55 (Inversion of subcapturing under platform environment).
If Γ is a platform environment and $\Gamma \vdash C <: D$, then either $C \subseteq D$ or $* \in D$.

Proof. By induction on the subcapturing relation. Case (sc-elem) trivially holds. Case (sc-set) holds by repeated IH. In case (sc-var), we have $C = \{x\}$ and $x : C'\, S \in \Gamma$. Since Γ is a platform environment, we have $C' = \{*\}$, which means that the other premise of (sc-var) is $\Gamma \vdash \{*\} <: D$. Since Γ is well-formed, $* \notin \mathrm{dom}(\Gamma)$, which means that we must have $* \in D$. □

Lemma A.56 (Strengthening of subcapturing). *If $\Gamma, \Gamma' \vdash C <: D$ and $C \subseteq \mathrm{dom}(\Gamma)$, then we must have $\Gamma \vdash C <: D$.*

Proof. First, we consider that if $* \in D$, we trivially have the desired goal. If $* \notin D$, we proceed by induction on the subcapturing relation. Case (sc-elem) trivially holds and case (sc-set) holds by repeated IH.

In case (sc-var), we have $C = \{x\}$, $x : C'\, S \in \Gamma, \Gamma'$. This implies that $\Gamma = \Gamma_1, x : C'\, S, \Gamma_2$ (as $x \notin \mathrm{dom}(\Gamma)$). Since Γ, Γ' is well-formed, we must have $\Gamma_1 \vdash C'$ **wf**. Since we already know $* \notin D$, then we must also have $* \notin C'$, which then leads to $C' \subseteq \mathrm{dom}(\Gamma_1)$. This in turn means that by IH and weakening we have $\Gamma \vdash C' <: D$, and since we also have $x : C'\, S \in \Gamma$, we can conclude by (sc-var). □

Then we will need to connect it to subcapturing, because the keys used to open boxes are supercaptures of the capability inside the box. We want:

Lemma A.57. *Let* Γ *be a platform environment,* $\Gamma \vdash \sigma \sim \Delta$ *and* $\Gamma, \Delta \vdash C_1 <: C_2$. *Then* $\mathrm{resolve}(\sigma, C_1) \subseteq \mathrm{resolve}(\sigma, C_2)$.

Proof. By induction on σ. If σ is empty, we have $\mathrm{resolve}(\sigma, C_1) = C_1$, likewise for C_2, and we can conclude by Lemma A.55.

Otherwise, $\sigma = \sigma'[\ \mathbf{let}\, x = v\, \mathbf{in}[\ ]\ ]$ and $\Delta = \Delta', x : D_x\, S_x$ for some S_x. Let $\theta = \mathrm{resolver}(\sigma)$. We proceed by induction on the subcapturing derivation. Case (sc-elem) trivially holds and case (sc-set) holds by repeated IH.

In case (sc-var), we have $C_1 = \{y\}$ and $y : D_y\, S_y \in \Gamma, \Delta$ for some S_y, and $\Gamma, \Delta \vdash D_y <: C_2$. We must have $\Gamma, \Delta' \vdash D_y$ **wf** and so we can strengthen subcapturing to $\Gamma, \Delta' \vdash D_y <: C_2$, which by IH gives us $\mathrm{resolver}(\sigma')(D_y) \subseteq \mathrm{resolver}(\sigma')(C_2)$. By definition we have $\theta = \mathrm{resolver}(\sigma) = \mathrm{resolver}(\sigma') \circ [x := \mathrm{cv}(v)]$. Since by well-formedness $x \notin D_y$, we now have:

$$\theta D_y \subseteq \theta C_2$$

By Lemma A.39 and Lemma A.48, we must have $\Gamma, \Delta \vdash \mathrm{cv}(v) <: D_y$. Since $\Gamma, \Delta \vdash \mathrm{cv}(v)$ **wf**, we can strengthen this to $\Gamma, \Delta \vdash \mathrm{cv}(v) <: D_y$. By outer IH this gives us $\mathrm{resolver}(\sigma')(\mathrm{cv}(v)) \subseteq \mathrm{resolver}(\sigma')(D_y)$. Since $x \notin \mathrm{cv}(v) \cup D_y$, we have:

$$\theta\, \mathrm{cv}(v) \subseteq \theta D_y$$

Which means we have $\theta\, \mathrm{cv}(v) \subseteq \theta C_2$ and we can conclude by $\theta\, \mathrm{cv}(v) = \theta\{x\}$, since:

$$\theta\{x\} = (\mathrm{resolver}(\sigma') \circ [x := \mathrm{cv}(v)])(\{x\}) = \mathrm{resolver}(\sigma')(\mathrm{cv}(v))$$
$$\theta\, \mathrm{cv}(v) = \mathrm{resolver}(\sigma')(\mathrm{cv}(v)) \quad (\text{since } x \notin \mathrm{cv}(v))$$

□

A.6.5 Core lemmas.

Lemma A.58 (Program authority preservation). *Let* $\Psi[\ t\ ]$ *be a well-typed program such that* $\Psi[\ t\ ] \longrightarrow \Psi[\ t'\ ]$. *Then* $\mathrm{cv}(t') \subseteq \mathrm{cv}(t)$.

Proof. By inspection of the reduction rule used.

Case (apply). Then $t = \sigma[\ e[\ x\, y\ ]\ ]$ and $t' = \sigma[\ e[\ [z := y]s\ ]\ ]$. Note that our goal is then:

$$\mathrm{resolver}(\sigma)(\mathrm{cv}(e) \cup \mathrm{cv}([z := y]s)) \quad \subseteq \quad \mathrm{resolver}(\sigma)(\mathrm{cv}(e) \cup \mathrm{cv}(x\, y))$$

If we have $x \in \mathrm{dom}(\Psi)$, then $\Psi(x) = \lambda(z : U)\, s$. By definition of platform, the lambda is closed and we have $\mathrm{fv}(s) \subseteq \{z\}$, which in turn means that $\mathrm{cv}([z := y]s) \subseteq \{y\} \subseteq \mathrm{cv}(x\, y)$. This satisfies our goal.

Otherwise, we have $x \in \mathrm{dom}(\sigma)$ and $\sigma(x) = \lambda(z : U)\, s$. Since x is bound in σ, we have $\mathrm{resolver}(\sigma)(\mathrm{cv}(\lambda(z : U)\, s) \cup \{y\}) \subseteq \mathrm{resolver}(\sigma)(\mathrm{cv}(x\, y)))$. Since $\mathrm{cv}([z := y]s) \subseteq \mathrm{cv}(\lambda(z : U)\, s) \cup \{y\}$, our goal is again satisfied.

Case (TAPPLY). Analogous reasoning.
Case (OPEN). Then $t = \sigma[\ e[\ C \multimap x\ ]\ ]$ and $t' = \sigma[\ e[\ z\ ]\ ]$. We must have $x \in \mathrm{dom}(\sigma)$ and $\sigma(x) = \Box\ z$, since all values bound in a platform must be closed and a box form cannot be closed. Since $\Psi[\ t\ ]$ is a well-typed program, there must exist some Γ, Δ such that Γ is a platform environment and $\vdash \Psi[\ \sigma\ ] \sim \Gamma, \Delta$.

If $z \in \mathrm{dom}(\sigma)$, then by Lemma A.39 and Lemma A.42 we have $\Gamma, \Delta \vdash z : C\ S_z$ for some S_z. By straightforward induction on the typing derivation, we then must have $\Gamma, \Delta \vdash \{z\} <: C$. Then by Lemma A.57 we have $\mathrm{resolver}(\sigma)(\{z\}) \subseteq \mathrm{resolver}(\sigma)(C)$, which lets us conclude by an argument similar to the (APPLY) case.

Otherwise, $z \in \mathrm{dom}(\Psi)$. Here we also have $\Gamma, \Delta \vdash \{z\} <: C$, which implies we must have $z \in C$, so we have $\mathrm{cv}(z) \subseteq \mathrm{cv}(C \multimap x)$ and can conclude by a similar argument as in the (APPLY) case.
Case (RENAME), (LIFT). The lemma is clearly true since these rules only shift subterms of t to create t'.

□

LEMMA A.59 (SINGLE-STEP PROGRAM TRACE PREDICTION). *Let $\Psi[\ t\ ]$ be a well-typed program such that $\Psi[\ t\ ] \longrightarrow \Psi[\ t'\ ]$. Then the primitive capabilities used during this reduction are a subset of $\mathrm{cv}(t)$.*

PROOF. By inspection of the reduction rule used.
Case (APPLY). Then $t = \sigma[\ e[\ x\ y\ ]\ ]$. If $x \in \mathrm{dom}(\sigma)$, the lemma vacuously holds. Otherwise, $x \in \mathrm{dom}(\Psi) \setminus \mathrm{dom}(\sigma)$. From the definition of cv, we have $\{x\} \setminus \mathrm{dom}(\sigma) \subseteq \mathrm{cv}(t)$. Since x is bound in Ψ, we then have $x \in \mathrm{cv}(t)$, which concludes.
Case (TAPPLY). Analogous reasoning.
Case (OPEN), (RENAME), (LIFT). Hold vacuously, since no capabilities are used by reducing using these rules.

□

LEMMA A.60 (PROGRAM TRACE PREDICTION). *Let $\Psi[\ t\ ]$ be a well-typed program such that $\Psi[\ t\ ] \longrightarrow^* \Psi[\ t'\ ]$. Then the primitive capabilities used during this reduction are a subset of $\mathrm{cv}(t)$.*

PROOF. By IH, Single-step program trace prediction and authority preservation. □

A.7 Avoidance

Here, we restate Lemma 3.3 and prove it.

LEMMA A.61. *Consider a term **let** $x = s$ **in** t in an environment Γ such that $E \vdash s : C_s\ U$ is the most specific typing for s in Γ and $\Gamma, x : C_s\ U \vdash t : T$ is the most specific typing for t in the context of the body of the let, namely $\Gamma, x : C_s\ U$. Let T' be constructed from T by replacing x with C_s in covariant capture set positions and by replacing x with the empty set in contravariant capture set positions. Then for every type V avoiding x such that $\Gamma, x : C_s\ S \vdash T <: V$, $\Gamma \vdash T' <: V$.*

PROOF. We will construct a subtyping derivation showing that $T' <: V$. Proceed by structural induction on the subtyping derivation for $T <: V$. Since T' has the same structure as T, most of the subtyping derivation carries over directly except for the subcapturing constraints in (CAPT).

In this case, in covariant positions, whenever we have $C_T <: C_V$ for a capture set C_T from T and a capture set C_V from V, we need to show that that $\vdash [x := C_s]C_T <: C_V$. Conversely, in contravariant positions, whenever we have $C_V <: C_T$, we need to show that $C_V <: [x := \{\}]C_T$. For the covariant case, since $x \in C_T$ but not in C_V, by inverting the subcapturing relation $C_T <: C_V$, we obtain $C_s <: C_V$. Hence $[x := C_s]C_T <: C_V$, as desired.

The more difficult case is the contravariant case, when we have $C_V <: C_T$. Here, however, we have that $C_V <: [x := \{\}]C_T$ by structural induction on the subcapturing derivation as x never occurs on the left hand side of the subcapturing relation as V avoids x. □