Thèse n° 9872

Formal Foundations of Capture Tracking

Présentée le 11 mars 2024

Faculté informatique et communications
Laboratoire de méthodes de programmation 1
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Aleksander Slawomir BORUCH-GRUSZECKI

Acceptée sur proposition du jury

Prof. O. N. A. Svensson, président du jury
Prof. M. Odersky, directeur de thèse
Prof. S. Drossopoulou, rapporteuse
Prof. C. S. Gordon, rapporteur
Prof. C. Pit-Claudel, rapporteur

École
polytechnique
fédérale
de Lausanne

2024

Do not go gentle into that good night,
Old age should burn and rave at close of day;
Rage, rage against the dying of the light.

— Dylan Thomas

Abstract

Type systems are a device for verifying properties of programs without running them. Many programming languages used in the industry have always had a type system, while others were initially created without a type system and later adopted one, when the advantages of doing so became apparent. Most such type systems stop at verifying that operations are invoked on appropriate operands, e.g., that we do not add characters and do not ask for the length of an integer. Still, verifying many desirable properties requires a type system that can describe what sort of operations a program might invoke. For instance, if a certain part of the program is interpreted during compilation, we would prefer this part to refrain from accessing operations whose result is non-deterministic, such as querying a database.

Naturally, various type systems which can verify such properties were proposed, with key areas including effect systems, resource ownership systems and object capabilities. The outward differences of these systems result in each of them being typically studied in the context of different categories of properties. Of course, ideally we would want to verify as many properties as possible, yet naively integrating different approaches in a single system does not provide good results in practice. Moreover, the industry has so far been slow to adopt the aforementioned systems, arguably because they are not yet sufficiently ergonomic and the costs of applying them outweigh their numerous advantages.

I describe Capture Tracking, an approach which views both effects and resources through the lens of capabilities tracked in types. The capability angle solves usability problems associated with effect and resource ownership systems and provides a uniform framework for verifying a wide range of properties. I present the key principles of Capture Tracking with the SCC system and discuss how to extend it with type polymorphism based on two other systems, $\mathsf{CF}_{<:}$ and $\mathsf{CC}_{<:\square}$. While doing so, I present extensions applying Capture Tracking to problems previously studied in the literature, including non-local returns, region-based memory management, and effect handlers. Finally I propose *gradual compartmentalization*, a technique for incrementally compartmentalizing a software application via object capabilities tracked in types, illustrating the practical applicability of Capture Tracking. The technique further relies on a concept of runtime authority enforcement. I present ModCC and GradCC as the formal foundations for the technique, extending $\mathsf{CC}_{<:\square}$ with mutable state, records, and a form of graduality specific to Capture Tracking which involves capture-unchecked values and dynamic enforcement of capability access restrictions.

Résumé

Les systèmes de types sont un outil permettant de vérifier certaines propriétés des programmes sans les exécuter. Beaucoup de langages de programmation utilisés dans l'industrie ont toujours été dotés d'un système de types, tandis que d'autres, bien qu'initialement conçus sans système de types, en ont adopté un lorsque les avantages sont devenus clairs. La plupart des systèmes de types se contentent de vérifier que chaque opération est appliquée à des opérandes appropriés, empêchant par exemple d'additionner des caractères ou de demander la longueur d'un entier. Cependant, il existe de nombreuses propriétés souhaitables dont la vérification exige un système de types capable de décrire quelles sortes d'opérations un programme est susceptible d'invoquer. Par exemple, si un sous-programme doit être interprété pendant la compilation, il est judicieux qu'il s'abstienne d'accéder à des opérations dont les résultats sont non déterministes, telles que les requêtes sur une base de données externe.
Naturellement, de nombreux systèmes de types dotés de telles capacités ont été proposés, dans des domaines clefs dont les systèmes à effets, à propriété des ressources, et à capacité des objets. À cause de leurs différences extérieures, ces systèmes sont généralement étudiés dans le contexte de catégories de propriétés distinctes. Si l'on souhaiterait idéalement vérifier autant de propriétés que possible, intégrer naïvement différentes approches dans un unique système ne donne en pratique pas de bons résultats. De plus, le secteur a jusqu'à présent été lent à adopter les systèmes mentionnés ci-dessus, vraisemblablement à cause de leur ergonomie insuffisante et des coûts associés à leur application, qui excèdent souvent leurs avantages, aussi nombreux soient-ils.
Dans cette thèse, je décris le Traçage des Captures (*Capture Tracking*), une approche unifiant les effets et les ressources comme capacités tracées par les types. Cette optique résoud les problèmes d'ergonomie associés aux systèmes à effets et à propriétés de ressources, tout en exposant un cadre uniforme pour vérifier un large spectre de propriétés. Je présente les principes clefs du Traçage des Captures avec le système SCC et traite de son extension avec du polymorphisme de types basé sur deux autres systèmes, $\mathsf{CF}_{<:}$ et $\mathsf{CC}_{<:\square}$. Ce faisant, je présente des extensions appliquant le Traçage des Captures à des problèmes étudiés précédemment dans la littérature, dont les retours non locaux, la gestion de la mémoire basée sur les régions, et les gestionnaires d'effets. Finalement, je propose la *compartimentalisation graduelle* (*gradual compartmentalization*), une méthode permettant de compartimenter incrémentalement une application logicielle *via* des capacités d'objets tracées par les types, illustrant ainsi l'usage du Traçage des Captures en pratique. Cette méthode repose sur un concept d'imposition d'une autorité à l'exécution. Je présente ModCC et GradCC comme les fondations formelles de cette

méthode, en étendant $\mathsf{CC}_{<:\Box}$ avec de l'état muable, des enregistrements, et une forme de gradation spécifique au Traçage des Captures qui repose sur des valeurs non assurées et une imposition dynamique des restrictions d'accès aux capacités.

Contents

1 Introduction

Type systems allow us to validate certain program properties statically, without running the program. A type system *is* a formal logic, one whose statements assign types to terms, which formally represent programs. This logic gives formal foundations for the type-checker in a compiler: the component which assigns a type to every expression in a program. In most type system, this type describes the "shape" of the values the expression may result in, i.e., what operations can be performed directly on said values: integers can be added, functions can be called, records have fields which can be accessed, etc. Most programming languages used in the industry feature type systems which stop at describing such “shapes”; doing so is enough to rule out many invalid programs and expressing certain shapes may already require intricate type system features.

Yet, many important properties can only be validated if we go beyond tracking shapes of values. Some properties require reasoning about what functionality may be accessed by a particular piece of code.

- In the context of compiler optimizations, certain optimizations are only valid for code that does not access side-effectful APIs (common subexpression elimination being possibly the simplest example).
- In a metaprogramming setting [Stucki 2023; Parreaux 2020], we similarly want to ensure that code evaluated at compile time does not access side-effectful APIs.
- In a concurrent setting, objects shared between threads should not allow accessing thread-unsafe APIs.
- In a setting without garbage collection, deallocated objects should not be accessed at all.
- In a security-sensitive setting, we may want to restrict untrusted objects from accessing privileged functionality, or we may want to ensure that private data is not accessed when computing public information.

Naturally, there are many approaches which facilitate this sort of reasoning, with some particularly salient areas being *effect systems* (e.g., Koka [Leijen 2014], Effekt [Brachthäuser et al. 2020b,a]; also see Cyclone [Grossman et al. 2002] and Verse [2023]), *resource ownership systems* (e.g., Rust [Rust 2023], Hylo, ownership types [see Clarke et al. 2013b,a; Mycroft and Voigt 2013], linearity [Wadler 1990], uniqueness [see Marshall et al. 2022]), and *capabilities* (e.g., E [Miller 2006], Wyvern [Melicher 2020], KeyKOS [Hardy 2023, 1985], sel4 [Klein et al. 2009]).

Clearly we would like to statically validate as many program properties as possible. We could certainly do so if we simply kept adding more and more features to the type system. Programming languages, however, are often said to have a "complexity budget": a cost-benefit analysis might tell us that the cost of learning and dealing with a particularly complex language outweighs the benefit of verifying more properties than what a simpler language would allow. Even before we get there, we may be overburdened by an incomprehensible, impossible to implement theory. Moreover, many existing systems already have some usability problems. Despite substantial amounts of research on systems for managing effects and resource ownership, neither approach is widely adopted in the industry. Arguably, the reason why is that the systems being proposed, despite having numerous significant advantages, are still insufficiently ergonomic and the costs of adopting them outweigh the benefits of doing so.

The core concept explored in my thesis is that object capabilities whose Capture is Tracked in types can be used to uniformly reason about both effects and resources, while also improving on the usability of both approaches.

1.1 Preliminaries

Before moving on, I briefly clarify some terminology used in the introduction. Each concept is introduced in greater detail as it becomes relevant to the subject at hand, and at the end of the thesis I give a more detailed outline of the background work.[1]

A type-and-effect system directly tracks the *effects* of terms. It is typically distinguished by a typing judgement that assigns an effect to terms as well as a type, although monads [see Wadler and Thiemann 2003] as realized in, e.g., Haskell, are an effect system as well for most intents and purposes. Early effect systems focused on tracking access to mutable state [Lucassen and Gifford 1988], but this was quickly extended to tracking almost any observable effect, such as throwing exceptions or potential divergence [Leijen 2014]. Algebraic effects and effect handlers [Plotkin and Power 2003; Plotkin and Pretnar 2013; Leijen 2016] are a particularly salient approach for extending a language with a powerful facility for defining custom effects.

A resource ownership system tracks *resources* by their identity or provenance (i.e., approximate identity) and restricts how they may be aliased; such systems often effectively enforce a particular topology on the heap. *Linear types* [Wadler 1990] are an early and particularly

[1]The outline is necessarily incomplete! The literature on ownership types alone is substantial enough to merit the existence of surveys *of surveys*.

restrictive example of such a system. Rust [Rust 2023] is perhaps the most widely known such system, especially if we take into account its industry mindshare.[2] The literature on *ownership types* features numerous examples of incredibly diverse resource ownership disciplines [see Clarke et al. 2013b,a; Mycroft and Voigt 2013]. Some of the most recent academic works on a resource ownership system include Hylo [Racordon et al. 2022], Gallifrey [Milano et al. 2022], and Verona's Reggio [Arvidsson et al. 2023a,b].

The object capability model declares that sensitive functionality may only be accessed by calling methods on *capabilities*, a special kind of objects. Capability-safe code should additionally have no *ambient authority*, i.e., an object should only be able to access pre-existing capabilities if it receives them from another object. This dissertation focuses on object capabilities in particular, but the term "capability" has a broader meaning. Object capabilities were first articulated as an idea (in a formal publication, at least) in Miller's seminal thesis on E [Miller 2006].

The above categories are neither exclusive nor exhaustive, for instance Wyvern features both object capabilities and an effect system.

1.2 Contributions

I present the formal foundations for Capture Tracking, an approach to tracking capabilities in types, based on a number of formal systems.

First I present SCC, the *Simple Capture Calculus*,[3] and discuss the fundamental notions on which the Capture Tracking approach is based. I conclude by showing the difficulties associated with extending SCC with universal type polymorphism.

Next I present $\mathsf{CF}_{<:}$ and $\mathsf{CC}_{<:\Box}$, two systems which take slightly but significantly different approaches to solving the type polymorphism problem. I provide a mechanized proof of soundness for $\mathsf{CF}_{<:}$ and a pen-and-paper proof of soundness for $\mathsf{CC}_{<:\Box}$. Afterwards I contrast the two approaches and discuss reasons why both of them are significant.

Finally I discuss Gradual Compartmentalization, an approach which uses Capture Tracking together with other mechanisms to allow gradually introducing object capabilities to existing codebases so that they can be compartmentalizated, as a validation that Capture Tracking is practical and applicable to real-world problems. I present Gradient, a hypothetical Gradual Compartmentalization extension to Scala, as well as GradCC, the foundational formal system behind Gradient. I validate Gradual Compartmentalization by migrating the standard Scala XML library to Gradient, and I provide a pen-and-paper proof of soundness for GradCC.

I conclude the thesis by discussing the background and the literature related to Capture

[2]Rust was voted the "most beloved" programming language in the annual global Stack Overflow survey *seven* times in a row.

[3]The calculus is *almost* simply-typed, except that it features variable-dependent types.

Tracking and Gradual Compartmentalization.

The $\mathsf{CF}_{<:}$ and $\mathsf{CC}_{<:\square}$ systems were presented in **Tracking Captured Variables in Types** [Boruch-Gruszecki et al. 2021] and **Capturing Types** [Boruch-Gruszecki et al. 2023]. The SCC calculus itself is technically a new contribution, although its presentation is based on the presentation of $\mathsf{CC}_{<:\square}$, at times rewritten in my own voice to suit the thesis better. The presented version of $\mathsf{CF}_{<:}$ was not published before; I point out the differences between it and the ArXiv preprint as it is presented. The entire presentation of $\mathsf{CF}_{<:}$ and the comparison between $\mathsf{CF}_{<:}$ and $\mathsf{CC}_{<:\square}$ were significantly rewritten for this thesis; large portions of the comparison are a new contribution.

The Gradual Compartmentalization work is based on an unpublished paper, a result of a collaboration between me and Adrien Ghosn, Mathias Payer, and Clément Pit-Claudel.

2 The Capture Tracking Approach

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: accessing mutable state, throwing an exception, accessing I/O functionality, suspending a computation (e.g., waiting for an event), using a continuation for control operations, or even non-termination.

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 criticized for being both too verbose and too rigid. The problem is not a lack of expressiveness, as systems have been proposed and implemented for many kinds of effects. Rather, the problem is the lack of usability and flexibility, with particular difficulties in describing polymorphism. This leads either to overly complex definitions, or to the necessity of duplicating large bodies of code.

Classical type-systematic approaches to tracking effects have a crucial flaw, 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 *transitively* calls, and traditional type-and-effect systems have no lightweight mechanism to describe this behavior. The standard approach is either manual specialization alongside specific effect classes,[1] which means large-scale code duplication, or quantifiers on all definitions alongside possible call graph edges to account for the possibility that some call target has an effect, which means large amounts of boilerplate in type signatures. 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 tracked in the type system [Craig et al. 2018; Miller 2006; Marino and Millstein 2009b; Gordon

[1]We can see this in Haskell, where applications such as GHC may be expressed in terms of a single monadic data type. Since monads (famously) do not naturally compose, Haskell's ecosystem features multiple solutions to making an application's code effect-polymorphic, including monad transformers and the tagless final representation.

2020; Liu 2016; Brachthäuser et al. 2020a; Osvald et al. 2016]. Capabilities exist in many forms, but for the purposes of this thesis I focus on object capabilities, i.e., capabilities which are objects accessible as regular program variables. For instance, consider the following two morally-equivalent formulations of a method in Scala[2].

```
def foo(): T throws E
def foo()(using 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`** type operator 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 thesis.

Aside: The link between the "effect" and the "capability" version of `f` can be made even more direct by means of context function types [Odersky et al. 2017]. It is embodied in the following definition of the **`throws`** type:

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

The context function type `CanThrow[E] ?=> T` represents functions from `CanThrow[E]` to `T` whose arguments are implicitly 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.

The crucial difference shows up once we consider what an effect-polymorphic function looks like. In the imaginary Scala-with-effects, the signature of the `map` function on `List` would look as follows.

```
class List[X]:
  def map[Y, E](f: X -> Y eff E): List[Y] eff E
```

Here, `X -> Y `**`eff`**` E` is hypothetical syntax for the type of a function that takes arguments of type `X`, produces results of type `Y` and causes effects `E`. In such a small example the signature may still look reasonable, but in practice the number of effect parameters quickly gets out of hand for more complicated higher-order functions. Indeed, many designers of programming languages with support for effect systems agree that programmers should ideally not be

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

confronted with explicit effect quantifiers [Brachthäuser et al. 2020a; Leijen 2017; Lindley et al. 2017]. This problem is especially acute in object-oriented languages, where practically every function is higher-order [Cook 2009].

The signature of `List#map` with Capture Tracking is instead as follows.

```
class List[X]:
  def map[Y, E](f: X => Y): List[Y]
```

Interestingly, it is exactly the same as the signature of `List#map` in current Scala, which does not track effects! The reason why this works is that `X => Y` is the type of impure functions, which may capture arbitrary capabilities; alongside it, we now also have a type of pure functions `X -> Y`, which may not close over any capabilities. Capture Tracking follows the capability style of thinking, which decomposes the effect space differently from classical effect systems. The `map` method by itself is understood to be pure, since it does not cause any effects using its own capabilities. When it is called with an impure closure as an argument, it *receives* the permission to use the closure and cause any effect the closure may cause. This does not require any additional annotation in the signature, since in the object capability model having a capability is the same as being able to use it. I argue that this mental model is closer to how most people think when writing programs: if a function takes a closure as an argument we naturally expect it to *use* the closure. It is the reverse situation, where the function *does not* immediately use the closure (and presumably stores it somewhere for later use, possibly in its own result), which is less common. Therefore, by following the capability model, Capture Tracking gains effect polymorphism for free.

This seems too good to be true and indeed there is a catch: since having a capability means being able to use it, it now becomes necessary to track where capabilities may go and how they may be reached; in particular we now need to reason about capabilities captured by closures.

To see why, consider that effect capabilities are often restricted to a particular scope and have a limited lifetime. In our capability-based exceptions example, a `CanThrow[E]` capability is created for the duration of a **`try`** block that catches exceptions `E`; this capability is only valid as long as the **`try`** block is executing, i.e., there is a handler for exceptions `E` on the stack. Figure 2.1 shows an example of capability-based checked exceptions; the left snippet shows the source syntax example, while the right snippet shows desugared code where the `CanThrow` capability is explicitly bound and passed as an argument. The **`try`** block has a **`catch`** clause with a case for `TooLarge` exceptions and accordingly, the block introduces a `CanThrow[TooLarge]` capability. Inside the block we map over a `List[Int]` using a lambda which captures the `CanThrow` capability; doing so eagerly creates a new `List[Int]`, making this usage of `CanThrow` well-scoped.

The following listing shows a slight variation of Figure 2.1 where the capability usage is ill-scoped. We map over an `Iterator` and not a `List`; since the elements of Scala `Iterator`-s are calculated lazily, by doing so we create a new `Iterator` which (indirectly) captures the

```
class TooLarge extends Exception

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

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

Figure 2.1: Exception handling: source (left) and elaborated code (right)

`CanThrow` capability. We return this `Iterator` from under the **`try`** block and call its `next` method, which may throw an unhandled exception.

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

A key question answered by Capture Tracking is how to rule out the snippet involving the lazy `map` of `Iterator` while still allowing the one with the strict `map` of `List`. This is far from a novel problem and there exists a large body of research which addresses it. Relevant techniques include linear types [Wadler 1990], rank 2 quantification [Launchbury and Sabry 1997], regions [Tofte and Talpin 1997; Grossman et al. 2002], uniqueness types [Barendsen and Smetsers 1996], ownership types [Clarke et al. 1998; Noble et al. 1998], and second class values [Osvald et al. 2016].

Speaking in broad strokes, the main issue with the existing approaches is their relatively high notational overhead, in particular when dealing with polymorphism. Additionally, such systems are almost universally *prescriptive*: they are aimed at restricting certain access patterns selected *a priori*, e.g., linear types only allow linear values to be accessed once. In contrast, Capture Tracking at its core is *descriptive*: it focuses first and foremost on *describing* what capabilities are captured by values of a given type.

This approach revolves around two interlinked concepts:

- A *capturing type* is of the form $T\,\hat{}\,\{c_1, c_2, \ldots, c_n\}$; it is a classical type T augmented with a *capture set* of capabilities $\{c_1, c_2, \ldots, c_n\}$.
- A *capability* is a variable (local or a parameter) whose type is a capturing type with a non-empty capture set. We also refer to such variables as *tracked*.

These types are based on the notion that *every* capability is derived from other, more sweeping capabilities, listed in the capture set of its type. Furthermore, Capture Tracking *posits* that there exists a *root capability* "**`cap`**", which is the "most sweeping" capability from which all others are derived.

In this chapter I present SCC, the *Simple Capture Calculus*, as a foundational system which allows reasoning about capture of capabilities and incorporates the key principles of Capture Tracking. The system intentionally does not support universal type polymorphism; at the end of this chapter we will see that combining Capture Tracking with universal type polymorphism results in subtle problems, which will be the topic of the next three chapters.

Additionally, whereas many of our motivating examples describe applications in effect checking, the formal treatment presented here avoids discussing any particular effects. In fact, the effect domains are intentionally kept open: the foundational system is intended to work with diverse effect extensions and so, to avoid enshrining any particular effect extension and privileging it over others, all particular applications are left out of the core operational semantics and left to extensions of the base system.

2.1 Key Aspects of Capture Tracking

I start by presenting and motivating the key elements of Capture Tracking. The examples are written in an experimental language extension of Scala 3 [Scala 2022b], and they apply to all the formalisms discussed in this dissertation.

2.1.1 Capability Hierarchy

An object is a capability if it allows accessing restricted functionality[3], either directly or indirectly. It follows that an object which captures a capability becomes a capability itself, which naturally organizes capabilities into a derivation hierarchy. Further, Capture Tracking *posits* that *all* capabilities are derived from other capabilities with the only exception being the root capability **`cap`**, from which all other capabilities are derived. **`cap`** is a type system fiction: it only exists to make the derivation hierarchy a tree rooted in **`cap`**. The following example illustrates the idea of capability derivation, and how it is integrated into Capture Tracking.

```
class Fs:
  … // methods for accessing the filesystem

class Logger(fs: Fs^{cap}):
```

[3]What functionality is "restricted", i.e., guarded by capabilities, depends on the context we are in. In a security-oriented setting like the E language, access to all system and hardware functionality is restricted, but exceptions may be thrown freely. In a setting where we control side effects with capabilities, throwing exceptions and accessing mutable state both should be guarded by a capability. Given a capability-based Rust-like memory management system, all heap-allocated objects would be capabilities, i.e., heap access would be restricted.

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

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

In the example, the `test` method takes as an argument `fs`, an object of type `Fs^{cap}` which allows accessing the file system. On the type level, `fs` is a capability since it has a non-empty capture set. The first step of `test` is to create an instance of `Logger`, initializing it with `fs`. The created object retains `fs` so that it may write logged messages to the disk. It is assigned to `log`, a local variable whose type is `Logger^{fs}`. Since `log` allows accessing the filesystem, it is a capability; since it retains `fs`, the capture set of its type is `{fs}`, i.e., it is *derived* from `fs`.

Next, `test` creates `xs`, a lazy list obtained from `LazyList.from(1)` by mapping consecutive integers while logging the operations. Since the elements of `xs` are computed lazily, it must retain a reference to `log` for its internal computations. Hence, the type of `xs` is `LazyList[Int]^{log}`: it is a capability derived from `log` which allows writing to the file system by logging messages.

Furthermore, capturing types come with a subtype relation where types with "smaller" capture sets are subtypes of types with larger sets. In our example, `xs` can also be typed as `LazyList[Int]^{fs}` and `LazyList[Int]^{cap}`. This is formally expressed with the *subcapturing* relation, which we will see shortly. If a type `T` does not have a capture set, we refer to it as *pure*, and it is a subtype of any capturing type that adds a capture set to `T`.

2.1.2 Function Types

The function type `A => B` stands for a function that can capture arbitrary capabilities. We call such functions *impure*. By contrast, the single arrow function type `A -> B` stands for a function that cannot capture any capabilities, i.e., a *pure* function. Syntactically, one can add a capture set after the arrow of an otherwise pure function. For instance, `A ->{c, d} B` would be a function that can at most capture capabilities `c` and `d`. It can be seen as a shorthand for the type `(A -> B)^{c, d}`.

The impure function type `A => B` is an alias for `A ->{cap} B`, i.e., impure functions are functions that can capture any capability.

2.1.3 Capture-Checking Closures

If a closure's body refers to capabilities, the set of those capabilities is used as the capture set for the type given to the closure, as illustrated by the following definition.

```
def test(fs: Fs^{cap}): String ->{fs} Unit =
  (x: String) => new Logger(fs).info(x)
```

Here, the body of `test` is a lambda that refers to the capability `fs`, which means that `fs` is retained by the lambda. Consequently, the type of the lambda is `String ->{fs} 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 the capabilities captured by the functions it calls. For instance, in

```
def test(fs: Fs^{cap}) =
  def f(x: String) = new Logger(fs).info(x)
  val g = (x: String) => f(x)
  g
```

the result of `test` has type `String ->{fs} Unit`, even though function `g` itself does not refer to `fs` and merely indirectly closes over it.

2.1.4 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 *subcaptures* C_2, or, swapping the operands, that C_2 *accounts for* C_1.

Subtyping extends as follows to capturing types:

- Pure types are subtypes of capturing types, i.e., we have $T <: T\,\hat{}\,C$ for any T and C.
- Smaller capture sets produce smaller types: $T_1\,\hat{}\,C_1 <: T_2\,\hat{}\,C_2$ 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 $\mathsf{fs} : \mathsf{Fs}\,\hat{}\,\{\mathbf{cap}\}, \mathsf{ct} : \mathsf{CanThrow}[\mathsf{Exception}]\,\hat{}\,\{\mathbf{cap}\}$ and $\mathsf{l} : \mathsf{Logger}\,\hat{}\,\{\mathbf{cap}\}$, we have the following subcapturing relationships.

$$\begin{array}{ccccc} \{l\} & <: & \{fs\} & <: & \{\mathbf{cap}\} \\ \{fs\} & <: & \{fs, ct\} & <: & \{\mathbf{cap}\} \\ \{ct\} & <: & \{fs, ct\} & <: & \{\mathbf{cap}\} \end{array}$$

The set consisting of the root capability `{cap}` accounts for every other capture set. This is a consequence of the fact that, ultimately, every capability is derived from `cap`.

2.1.5 Escape Checking

Capture sets describe the capabilities captured by an object. We can use this to ensure that if a capability is scoped, it is not accessed outside of its scope. The idea is that when an object leaves the scope of some capability, we make sure all the capabilities in the capture set of its type are bound outside of the scope; in a sense, we check that all the capabilities reachable through the object are still accessible. Then, we introduce scoped capabilities with a type whose capture set contains `cap`, so that they *aren't* accounted for by anything bound outside of their scope: the only thing that accounts for them is `cap`, which is never bound anywhere. In other words, to ensure that access to particular capabilities is well-scoped, it suffices to (1) introduce them as capturing `cap` and (2) ensure that all objects leaving some capability's scope can be given a type with a capture set which does not contain `cap`.

It turns out to be convenient to also restrict type variables from ranging over types capturing `cap`, for two reasons. First, it allows abstracting over the types of objects leaving the scope of some capability. Second, it allows user-defined scoped capabilities, as illustrated by the following example, based on the the *try-with-resources* pattern,

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

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

The `usingFile` method runs its argument `op` on a freshly created file, closes the file, and returns the operation's result. The method enables an effect (writing to a particular file) with a limited validity (until the file handle is closed). Function `good` calls `usingFile` with an operation which 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 precise issue is that a write happens when the returned function is called `later(1)`, after the file handle has already been closed. We can see this possibility in the type system: the closure argument to `usingFile` in `fail` has the type `(os: OutputStream^{cap}) -> (() ->{os} Unit)`, informing us that the closure's result may capture its argument. Additionally, `later` has the type `() -> {cap} Unit`, which tells us it may capture arbitrary capabilities.

Capture Tracking allows allowing the first usage and rejecting the second by giving the capture set `{cap}` to the type of the output stream passed to `op`. The Scala implementation of capture checking rejects the second usage with an error message that the result of `usingFile` leaks `f`. The error occurs since the implementation disallows instantiating type variables with types whose capture sets contain `cap`, which happens in body of `fail`, in the call to `usingFile`.

To ensure that capability access is well-scoped, the type system needs to apply the same general principle to every way an object can leave some capability's scope. Another commonplace way for an object to do so would be by being stored in mutable state, for example like this.

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

Here, the closure passed to `usingFile` attempts to leak the scoped capability by creating a closure capturing it and assigning the closure to a mutable variable, `loophole`.[4] Disallowing mutable state from having a type whose capture set contains `cap` prevents such issues.

A very similar example further illustrates the usefulness of disallowing type variables from ranging over types capturing `cap`,

```
class Cell[A](a: A):
  var state: A = a

val loophole = new Cell[() ->{cap} Unit]( () => () )
usingFile("tryEscape", f =>
  loophole.state = () => f.write(0)
)
loophole.state()
```

[4]The `loophole` variable stores closures which take no arguments and which return `Unit`. Scala supports multi-argument closures, unlike the formal systems we will see later.

This example is very similar to the previous one, except that now `loophole` is an immutable variable, a `Cell` instance. Class `Cell` has a mutable field, and it abstracts over the type of this mutable state. The instance of `Cell` assigned to `loophole` allows storing closures capturing arbitrary capabilities. Once again, we can exploit this to access out-of-scope capabilities like in the previous example. Since it is clearly desireable to define classes which abstract over the type of their mutable state, like `Cell`, disallowing type variables from ranging over types capturing **`cap`** simultaneously prevents scoping issues involving such classes *and* supports user-defined scoped capabilities, as illustrated with `usingFile`.

While the system presented in this chapter does not feature scoped capabilities or mutable state, the later systems do; in particular, scoped capabilities are present in systems in Chapter 3 and Chapter 4 and mutable state is featured by systems in Chapter 3 and Chapter 6.

2.2 The SCC Calculus

Figure 2.2 shows the syntax of SCC, which stands for Simple Capture Calculus and can be succintly explained as a dependently-typed variant of $\lambda_{<:}$ with capturing types. The notation $\overline{E_i}^{i}$ denotes a *syntactic repetition* of a non-negative number of syntax forms E_i.

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.

Variable	$x, y, z, \mathbf{cap}$		
Value	v, w	$::=$	$\lambda(x : T)\,t$
Answer	a	$::=$	$v \mid x$
Term	s, t	$::=$	$a \mid x\,y \mid \mathbf{let}\,x = s\,\mathbf{in}\,t$
Shape Type	S	$::=$	$\top \mid \forall(x : U)\,T$
Type	T, U	$::=$	$S \mid S\,\hat{}\,C$
Capture Set	C	$::=$	$\{\overline{x}\}$
Typing Context	Γ, Δ	$::=$	$\emptyset \mid \Gamma, x : T \qquad \mathbf{if}\ x \neq \mathbf{cap}$

Figure 2.2: SCC syntax.

Monadic normal form. The term structure of SCC only allows variables as operands in application. This approach is not a restriction on the surface syntax and does not incur a loss of expresiveness, since classical-form terms can be easily elaborated into our formalism: a general application $t_1\, t_2$ can be expressed as $\textbf{let}\, x_1 = t_1 \,\textbf{in}\, \textbf{let}\, x_2 = t_2 \,\textbf{in}\, x_1\, x_2$. This form is a convenient way to formulate a formal system with variable-dependent types, since we essentially obtain a name for every significant value. 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.[5] Programs may be written in the usual direct style. They can be elaborated during type-checking as necessary, as the Scala compiler does.

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 $S\,\hat{}\,\{\overline{x}\}$. "^" has a higher precedence than $\forall$, for instance $\forall(x : S)\,\{C\}\,T$ is read as $\forall(x : S)\,(\{C\}\,T)$. Shape types comprise the usual type constructors, which for SCC are simply function types. We freely use shape types in place of types, assuming the equivalence $S\,\hat{}\,\{\} \equiv S$.

Capture sets. Capture sets C are finite sets of variables of the form $\{\overline{x}\}$. The root capability **cap** is a special variable that can appear in capture sets, but cannot be bound in Γ.

STLC with subtyping. SCC is based on the Simply Typed Lambda Calculus extended with subtyping, a principal form of type polymorphism. 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.

The only form of term-dependency in SCC is related to capture sets in types. If we omit capture sets, the calculus is equivalent to standard $\lambda_{<:}$, despite the differences in the syntactic form of terms. In the figures we highlight essential additions w.r.t. $\lambda_{<:}$ with a grey background.

SCC is intentionally meant to be a small, canonical core calculus which does not include high-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.

Additionally, SCC intentionally leaves out another fundamental form of type polymorphism,

[5]Monadic Normal Form [Hatcliff and Danvy 1994] is a slight generalization of the rather more widely known A-Normal Form (ANF) [Sabry and Felleisen 1993]. MNF allows arbitrary nesting of let expressions; systems in this dissertation use a variant of MNF where applications are over variables instead of values.

namely System F universal type polymorphism. There are subtle issues associated with extending SCC with universal type polymorphism, and they will be studied in later chapters (Chapter 3 and Chapter 4)

Many different systems can be built on SCC, extending it with various constructs to organize code and data on a higher level. In fact, in later chapters we will see two ways of extending it with universal type polymorphism, an extension with records and mutable state (Section 6.4) as well as diverse effect extensions (Section 3.3).

2.2.1 Preliminaries

I write $C \setminus x$ as a shorthand for subtraction of capture sets $C \setminus \{x\}$.

Term substitutions $[x := y]\, t$ simultaneously replace the variable in both term and capture set element positions.

The free variables $\mathrm{fv}(t)$ of a term t only include variables in term position; they do not include variables which occur in types (concretely, in capture sets) appearing in t.

2.2.2 Subcapturing

Subcapturing establishes a preorder relation on capture sets that gets propagated to types (Figure 2.3). The relation is defined by three rules. The first two, (SC-SET) and (SC-ELEM), establish that subcapturing extends the subset relationship.

The third 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 $S\,\hat{}\,C$ 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.

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 $x : S\,\hat{}\,\{\}$, then by (SC-VAR) we have $\{x\} <: \{\}$. Even if x occurs in a term, a capture set with x in it is equivalent to a capture set without x, i.e., either one subcaptures the other. Hence, x can safely be dropped without affecting subtyping or typing.

While rules (SC-SET) and (SC-ELEM) mean that we have $C <: C'$ if C is a subset of C', the reverse is not necessarily true. For instance, we can derive the following relationship via (SC-VAR) (where Proc is the procedure type Unit → Unit).

$$x : \mathsf{Proc}\,\hat{}\,\{\mathbf{cap}\}, y : \mathsf{Proc}\,\hat{}\,x \vdash \{y\} <: \{x\}$$

The intuitive reason why we can do so is that y can capture *no more* than x. However, we

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

$$\textsc{sc-elem}\ \frac{x \in C}{\Gamma \vdash \{x\} <: C} \qquad \textsc{sc-set}\ \frac{\overline{\Gamma \vdash \{x_i\} <: C}^{\,i}}{\Gamma \vdash \{\overline{x_i}^{\,i}\} <: C} \qquad \textsc{sc-var}\ \frac{x : S\,\hat{}\,C' \in \Gamma \qquad \Gamma \vdash C' <: C}{\Gamma \vdash \{x\} <: C}$$

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

$$\textsc{top}\ \Gamma \vdash S <: \top \qquad \textsc{refl}\ \Gamma \vdash T <: T \qquad \textsc{trans}\ \frac{\Gamma \vdash T_1 <: T_2 \quad \Gamma \vdash T_2 <: T_3 \quad \Gamma \vdash T_2\ \textbf{wf}}{\Gamma \vdash T_1 <: T_3}$$

$$\textsc{fun}\ \frac{\Gamma \vdash U_2 <: U_1 \qquad \Gamma, x : U_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x : U_1)\, T_1 <: \forall(x : U_2)\, T_2} \qquad \textsc{capt}\ \frac{\Gamma \vdash C_1 <: C_2 \qquad \Gamma \vdash S_1 <: S_2}{\Gamma \vdash S_1\,\hat{}\,C_1 <: S_2\,\hat{}\,C_2}$$

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

$$\textsc{var}\ \frac{x : S\,\hat{}\ C \in \Gamma}{\Gamma \vdash x : S\,\hat{}\ \{x\}} \qquad \textsc{sub}\ \frac{\Gamma \vdash t : T \qquad \Gamma \vdash T <: U \qquad \Gamma \vdash U\ \textbf{wf}}{\Gamma \vdash t : U}$$

$$\textsc{let}\ \frac{\Gamma \vdash u : U \qquad \Gamma, x : U \vdash t : T \qquad x \notin \mathrm{fv}(T)}{\Gamma \vdash \textbf{let}\, x = u\, \textbf{in}\, t : T} \qquad \textsc{abs}\ \frac{\Gamma, x : U \vdash t : T \qquad \Gamma \vdash U\ \textbf{wf}}{\Gamma \vdash \lambda(x : U)\, t : (\forall(x : U)\ T)\,\hat{}\ \mathrm{fv}(t) \backslash x}$$

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

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

(APPLY)	$\sigma[\,\eta[\,x\,y\,]\,]$	$\longrightarrow$	$\sigma[\,\eta[\,[z := y]t\,]\,]$	**if** $\sigma(x) = \lambda(z : T)\, t$
(RENAME)	$\sigma[\,\eta[\,\textbf{let}\, x = y\, \textbf{in}\, t\,]\,]$	$\longrightarrow$	$\sigma[\,\eta[\,[x := y]t\,]\,]$	
(LIFT)	$\sigma[\,\eta[\,\textbf{let}\, x = v\, \textbf{in}\, t\,]\,]$	$\longrightarrow$	$\sigma[\,\textbf{let}\, x = v\, \textbf{in}\, \eta[\,t\,]\,]$	**if** $\eta \neq [\,]$

Store context	σ	$::=$	$[\,] \mid \textbf{let}\, x = v\, \textbf{in}\, \sigma$
Eval context	η	$::=$	$[\,] \mid \textbf{let}\, x = \eta\, \textbf{in}\, t$

Figure 2.3: SCC typing and evaluation rules.

cannot derive $\{x\} <: \{y\}$, since arguments passed for y may in fact capture *less* than x, e.g., they could be pure.

$\{\mathbf{cap}\}$ and $\{\}$ are respectively the top and bottom capture sets, even though there are no explicit subcapturing rules for either of them. (This only holds for well-formed capture sets, which I define later.)

Proposition 2.1. *If C is well-formed in Γ, then $\Gamma \vdash \{\} <: C <: \{\mathbf{cap}\}$.*

It is perhaps more intuitive that $\{\}$ is the bottom capture set, since it is a subset of all other sets. It is slightly trickier to see that $\{\mathbf{cap}\}$ is the top capture set. This property is a consequence of having to bind every capability at a capture set which describes the capabilities captured by the variable. In order to bind the first capability in Γ, we *must* bind it with a capture set mentioning **cap**; there is no other variable we could possibly use. All variables bound before the first capability must be untracked, meaning that for every such variable y we will have $\{\} <: \{y\} <: \{\mathbf{cap}\}$ (see Section 2.3 reg. the proof).

We can show that transitivity and reflexivity are admissible, which establishes the following proposition.

Proposition 2.2. *The subcapturing relation $\Gamma \vdash \_ <: \_$ is a preorder.*

See Section 2.3 regarding the proof.

2.2.3 Subtyping

The subtyping rules of SCC (Figure 2.3) are very similar to those of $\lambda_{<:}$, with the only significant addition being a single rule for capturing types. Note that as $S \equiv S\,\hat{}\,\{\}$, both transitivity and reflexivity apply to shape types as well. Rule (CAPT) allows comparing types that have capture sets, where smaller capture sets lead to smaller types.

2.2.4 Typing

The typing rules of SCC (Figure 2.3) also differ from $\lambda_{<:}$ only to account for capture sets.

Rule (VAR) refines the capture sets at which variables are typed. If x is declared with type $S\,\hat{}\,C$, then x itself is typed at capture set $\{x\}$ instead of C. The capture set $\{x\}$ is more specific than C, in the subcapturing sense, and we can recover the capture set C through subsumption. This rule ensures that lambdas which immediately return their arguments have an appropriately polymorphic type, as in the following example: `choose` returns either its second argument x or its third argument y and the capture set of its result is $\{x, y\}$.

$$\begin{aligned}\mathsf{choose} &: \forall(b : \mathsf{Bool})\,\forall(x : \mathsf{Proc}\,\hat{}\,\{\mathbf{cap}\})\,\forall(y : \mathsf{Proc}\,\hat{}\,\{\mathbf{cap}\}) \\ &\qquad \mathsf{Proc}\,\hat{}\,\{x, y\} \\ \mathsf{choose} &= \lambda(b : \mathsf{Bool})\,\lambda(x : \mathsf{Proc}\,\hat{}\,\{\mathbf{cap}\})\,\lambda(y : \mathsf{Proc}\,\hat{}\,\{\mathbf{cap}\}) \\ &\qquad \textbf{if}\; b\; \textbf{then}\; x\; \textbf{else}\; y\end{aligned}$$

Rule (ABS) assigns capture sets to lambda-abstractions based on the free variables of the term. Untracked variables can be removed from this set via subsumption and rule (SC-VAR); observe that b is absent from the second and third lambda's capture sets.

The (APP) rule must replace references to the lambda's parameter in its result type; it does so with the argument passed to the lambda. This is possible since arguments are guaranteed to be variables. The lambda's capture set C is disregarded, reflecting the principle that having access to a capability is the same as being able to use it, regardless of what capabilities it may internally depend on.

Avoidance. As is usual in dependent type systems, rule (LET) has a side condition which ensures that the let-bound variable x does not appear 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

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

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

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

Proposition 2.3. *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 (w.r.t. subtyping) type* T_3 *such that* $T_2 <: T_3$ *and* $x \notin \mathrm{fv}(T_3)$.

See Section 2.3 regarding the proof.

2.2.5 Well-Formedness

Well-formedness $\Gamma \vdash T\; \mathbf{wf}$ (Figure 2.4) is as expected, in the sense that the free variables in types and terms must be defined in the environment, except that capturing types may mention the root capability **cap** in their capture sets.

Well-formedness $\boxed{\Gamma \vdash C\ \mathbf{wf}}$ $\boxed{\Gamma \vdash T\ \mathbf{wf}}$

$$\textsc{wf-cset}\ \frac{C \subseteq \mathrm{dom}(\Gamma) \cup \{\mathbf{cap}\}}{\Gamma \vdash C\ \mathbf{wf}} \qquad \textsc{wf-top}\ \Gamma \vdash \top\ \mathbf{wf} \qquad \textsc{wf-fun}\ \frac{\Gamma \vdash U\ \mathbf{wf} \qquad \Gamma, x : U \vdash T\ \mathbf{wf}}{\Gamma \vdash \forall(x : U) T\ \mathbf{wf}} \qquad \textsc{wf-capt}\ \frac{\Gamma \vdash C\ \mathbf{wf} \qquad \Gamma \vdash S\ \mathbf{wf}}{\Gamma \vdash S\,\hat{}\,C\ \mathbf{wf}}$$

Figure 2.4: SCC well-formedness rules.

2.2.6 Reduction

The operational semantics of SCC are defined by a small-step reduction relation. This relation is quite different from usual reduction via term substitution, since substituting values for variables would break the MNF form of SCC terms. Instead, we reduce the right hand sides of let-bound variables in place and look up what values are variables bound to in the context surrounding 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 $\mathbf{let}\,x = t_1\ \mathbf{in}\ t_2$. By contrast, a store context σ puts the focus on the following term t_2 and requires that t_1 is evaluated.

There are only three reduction rules. The first one, (APPLY), rewrites applications. It 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** forms in redex position. If the right hand side of the form is a variable, the **let** form gets expanded out by renaming the bound variable using (RENAME). If it is a value, the form gets lifted out into the store context using (LIFT).

Proposition 2.4. *Evaluation is deterministic. If $t \longrightarrow u_1$ and $t \longrightarrow u_2$, then $u_1 = u_2$.*

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

2.3 Metatheory

I now describe the metatheoretic properties of SCC. These properties are, essentially, restatements of the (chronologically earlier) developments for $\mathsf{CC}_{<:\Box}$ (Chapter 4), since SCC itself is the monomorphic fragment of $\mathsf{CC}_{<:\Box}$. Properties analogous to the ones presented in this chapter were proven for $\mathsf{CC}_{<:\Box}$ with a classical pen-and-paper proof. Since SCC merely removes certain forms and their typing rules[6] from $\mathsf{CC}_{<:\Box}$, the $\mathsf{CC}_{<:\Box}$ proofs are also applicable

[6]Concretely, SCC is $\mathsf{CC}_{<:\Box}$ without universal type polymorphism and *boxes*, as we will see later (Chapter 4).

to SCC properties merely by skipping certain cases during induction. The proofs for $\mathsf{CC}_{<:\Box}$ are included in the appendix.

The metatheory of SCC was developed following the Barendregt convention: we only consider typing contexts where all variables are unique, i.e., for all contexts of the form $\Gamma, x : T$ we have $x \notin \mathrm{dom}(\Gamma)$.

The Progress and Preservation Theorems both depend on a notion of a typing context *matching* a store context (Figure 2.5).

$$\frac{\Gamma \vdash v : T \qquad \Gamma, x : T \vdash \sigma \sim \Delta}{\Gamma \vdash \textbf{let}\, x = v\, \textbf{in}\, \sigma \sim x : T, \Delta} \qquad\qquad \Gamma \vdash [\,] \sim \cdot$$

Figure 2.5: Matching environment $\boxed{\Gamma \vdash \sigma \sim \Delta}$

Intuitively, having $\Gamma \vdash \sigma \sim \Delta$ means that σ is well-typed in Γ if we use Δ as the types of the bindings. The following four lemmas illustrate how store and evaluation contexts interact with typing.

Definition 2.1 (Evaluation Context Typing)**.** *Evaluation context η can be typed at $U \Rightarrow T$ in Γ, written $\Gamma \vdash \eta : U \Rightarrow T$, iff for all t such that $\Gamma \vdash t : U$, we have $\Gamma \vdash \eta[\, t\,] : T$.*

Lemma 2.1 (Evaluation Context Typing Inversion)**.** *$\Gamma \vdash \eta[\, s\,] : T$ implies that for some U we have $\Gamma \vdash e : U \Rightarrow T$ and $\Gamma \vdash s : U$.*

Lemma 2.2 (Evaluation Context Reification)**.** *If both $\Gamma \vdash \eta : U \Rightarrow T$ and $\Gamma \vdash s : U$, then $\Gamma \vdash \eta[\, s\,] : T$.*

Lemma 2.3 (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 2.4 (Store Context Reification)**.** *If $\Gamma, \Delta \vdash t : T$ and $\Gamma \vdash \sigma \sim \Delta$, then also $\Gamma \vdash \sigma[\, t\,] : T$.*

We can now proceed to our main soundness theorems. Their statements differ slightly from $\lambda_{<:}$, as we need to account for MNF. First, I state the Preservation Theorem.

Theorem 2.1 (Preservation)**.** *If we have $\Gamma \vdash \sigma \sim \Delta$ and $\Gamma, \Delta \vdash t : T$, then $\sigma[\, t\,] \longrightarrow \sigma[\, t'\,]$ implies that $\Gamma, \Delta \vdash t' : T$.*

The statement of the Preservation Theorem captures the idea that the significant type to preserve is the one assigned to the term under the store, not the one assigned to the entire reduced term. This mirrors how the Preservation Theorem would be stated for a more classical system which reduces store-term pairs.

Next I state the Progress Theorem, which needs an auxilliary definition first.

Definition 2.2 (Proper Configuration)**.** *A term form $\sigma[\ t\ ]$ is a* proper configuration *if t is not of the form* $\textbf{let}\, x = v\, \textbf{in}\, t'$.

Theorem 2.2 (Progress)**.** *If $\vdash \sigma[\ t\ ] : T$ and $\sigma[\ t\ ]$ is a proper configuration, then either t is an answer a, or $\sigma[\ t\ ] \longrightarrow \sigma[\ t'\ ]$ for some t'.*

Capture sets and captured variables

The SCC typing rules use fv 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 terms are typed at?

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 2.5 (Capture Prediction for Answers)**.** *If $\Gamma \vdash \sigma[\ a\ ] : S\,\hat{}\,C$, then $\Gamma \vdash \mathrm{fv}(\sigma[\ a\ ]) <: C$.*

If we start with an unreduced term $\sigma[\ t\ ]$, the situation becomes more intricate. Such a term 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 unreduced terms under a store context predict variables that will be captured by the answer those terms reduce to, very much like types assigned to unreduced terms by classical type systems describe the results of terms.

Lemma 2.6 (Capture Prediction for Terms)**.** *Let $\vdash \sigma \sim \Delta$ and $\Delta \vdash t : S\,\hat{}\,C$. Then $\sigma[\ t\ ] \longrightarrow^* \sigma[\ \sigma'[\ a\ ]\ ]$ implies that $\Delta \vdash \mathrm{fv}(\sigma'[\ a\ ]) <: C$.*

2.4 Conclusion

SCC is a minimal formal system which shows the key aspects behind Capture Tracking. So far, I have intentionally avoided paying attention to the problem of universal type polymorphism. To put it briefly, naively extending SCC with such polymorphism quickly violates the Capture Prediction Lemmas. Concretely, assume that we add type variables as a new shape type form together with type abstraction term and type forms; type variables may range over arbitrary types. Then it becomes possible to hide the capture of a function which may capture arbitrary capabilities simply by going through a type abstraction first, as the following example shows.

$$
\begin{aligned}
&\vdash \Lambda[F <: (\mathsf{Unit} \to \mathsf{Unit})\,\hat{}\,\{\mathbf{cap}\}]\,\lambda(f : F)\,\lambda(x : \mathsf{Unit})\,\mathbf{let}\,y = f\,()\,\mathbf{in}\,()\\
&: \forall[F <: (\mathsf{Unit} \to \mathsf{Unit})\,\hat{}\,\{\mathbf{cap}\}]\,\forall(f : F)\,\forall(x : \mathsf{Unit})\,\mathsf{Unit}
\end{aligned}
$$

Observe that the last closure has a type with an empty capture set despite capturing f, itself a closure which may capture arbitrary capabilities. The crux of the issue is that the naive extension treats type variables like F as pure types since they lack a capture set, even though there *is* a capture set hidden "under" the type variable. Clearly, we need to be just a bit more sophisticated when extending SCC with universal type polymorphism. There are two avenues we might take: either make the overall system aware that there are capture sets "under" type variables, or make type variables range over pure types only. These avenues correspond to the approaches I will present in the following two chapters.

3 Impure Type Variables: $\mathsf{CF}_{<:}$

In this chapter I present $\mathsf{CF}_{<:}$, one of the results of a long collaboration with Martin Odersky, Jonathan Brachthäuser, Ondřej Lhoták, and Edward Lee. To resolve the issues that arise when extending SCC with type polymorphism, we developed and evaluated a number of formal systems where type variables can range over arbitrary types, with $\mathsf{CF}_{<:}$ being the last entry in that line of systems. (We explored one other approach to type polymorphism, which I discuss in the next chapter.) A system very similar to $\mathsf{CF}_{<:}$ was described in an ArXiv preprint [Boruch-Gruszecki et al. 2021]. In contrast to that one, the system presented in this thesis allows type variables to appear in capture sets. Doing so has subtle but important consequences, some of which are discusses in Chapter 5.

The rest of this chapter is organized as follows. Section 3.1 presents $\mathsf{CF}_{<:}$ and Section 3.2 presents its metatheory. Then, Section 3.3 evaluates the presented formalism, first in terms of types assigned to common data structures (Section 3.3.1) and then by demonstrating its applicability to problems previously studied in the literature (Section 3.3.2, Section 3.3.3, Section 3.3.4, Section 3.3.5).

3.1 The $\mathsf{CF}_{<:}$ Calculus

3.1.1 Syntax of Terms and Types

Figure 3.1 defines the syntax of $\mathsf{CF}_{<:}$, which can be concisely explained as a dependently-typed version of System $\mathsf{F}_{<:}$ with capturing types. I present it by contrasting it with SCC.

Classical term form. $\mathsf{CF}_{<:}$ is chronologically older than SCC, and unlike SCC it does not restrict its terms to MNF, i.e., application operands can be arbitrary terms. As a result of that, it needs to introduce *other* (arguably worse) restrictions, which are incorporated into its well-formedness rules (Section 3.1.7) and explicit well-formedness premises in (ABS) and (T-ABS). Note that the atypical definition of well-formedness is necessary *only* because the system is not in MNF. $\mathsf{CF}_{<:}$ could be presented in MNF, at which point the system would be a

Variable	$x, y, z, \mathbf{cap}$		
Type Variable	X, Y, Z		
Value	v, w	$::=$	$\lambda(x : T)\, t \mid \Lambda[X <: T]\, t$
Term	s, t	$::=$	$v \mid x \mid t\, t \mid t\,[T]$
Pretype	U, V, W	$::=$	$\top \mid \forall(x : T)\, T \mid \forall[X <: T]\, T$
Type	R, S, T	$::=$	$X \mid U\,\hat{}\,C$
Capture Set	C	$::=$	$\{\overline{c}\}$
Capture	c	$::=$	$x \mid X$
Typing Context	Γ, Δ	$::=$	$\emptyset \mid \Gamma, x : T \mid \Gamma, X <: T$ **if** $x \neq \mathbf{cap}$

Figure 3.1: $\mathsf{CF}_{<:}$ syntax.

direct extension of SCC.

Types and Pretypes. Where SCC distinguishes types and shape types, $\mathsf{CF}_{<:}$ makes a sharp distinction between types T and *pretypes* U. Types T have only two forms: capturing types $U\,\hat{}\,C$ and type variables X. Type variables stand for a complete type $U\,\hat{}\,C$, i.e., both a capture set C and a pretype U. Pretypes U are classical types like function types and type abstraction types: they describe the "shape" of the value. Distinguishing between types and pretypes syntactically prevents type variables from being given an additional capture set.

By convention empty capture sets can be omitted in writing, making pretypes U appear where a capturing type $U\,\hat{}$ is grammatically expected. In practice, this has the same effect as SCC positing an equivalence between shape types S and capturing types with an empty capture set $S\,\hat{}\,\{\}$.

Capture Sets. $\mathsf{CF}_{<:}$ capture sets $\{\overline{c}\}$ are finite sets of *captures*. A capture c is either a term variable x or a type variable X, i.e., $\mathsf{CF}_{<:}$ capture sets extend SCC capture sets, since they can contain type variables. Intuitively, a type variable in a capture position stands for the capture set of the type it will be instantiated to.

Since type variables may appear in capture sets, every proper type T has a capture set $\mathrm{cs}(T)$ associated with it: the capture set C in case of capturing types $U\,\hat{}\,C$, and the type variable $\{X\}$ in case of type variables X.

3.1.2 Preliminaries

Definition 3.1 (Type capture set)**.** *The capture set* $\mathrm{cs}(T)$ *of a type* T *is defined as follows.*

$$\mathrm{cs}(U\,\hat{}\,C) = C \qquad\qquad \mathrm{cs}(X) = \{X\}$$

Evaluation $\boxed{t \longrightarrow t}$

$$\textsc{beta-v}\quad (\lambda(x : T)\ t)\ v \longrightarrow [x := v,\ x := \mathrm{fv}(v)\,]\,t \qquad \textsc{beta-T}\quad (\Lambda[x <: S]\ t)\,[T] \longrightarrow [x := T]\,t \qquad \textsc{context}\ \frac{t_1 \longrightarrow t_2}{\mathsf{E}[t_1] \longrightarrow \mathsf{E}[t_2]}$$

$$\mathsf{E} ::= [\,] \ \mid\ \mathsf{E}\,t \ \mid\ \mathsf{E}\,[T] \ \mid\ v\,\mathsf{E}$$

Figure 3.2: Small step operational semantics of the $\mathsf{CF}_{<:}$ calculus.

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

$$\textsc{sc-elem}\ \frac{x \in C}{\Gamma \vdash \{x\} <: C} \qquad \textsc{sc-set}\ \frac{\overline{\Gamma \vdash \{x_i\} <: C}^{\,i}}{\Gamma \vdash \{\overline{x_i}^{\,i}\} <: C} \qquad \textsc{sc-var}\ \frac{x : T \in \Gamma \quad \Gamma \vdash \mathrm{cs}(T) <: C}{\Gamma \vdash \{x\} <: C} \qquad \textsc{sc-tvar}\ \frac{X <: T \in \Gamma \quad \Gamma \vdash \mathrm{cs}(T) <: C}{\Gamma \vdash \{X\} <: C}$$

Figure 3.3: $\mathsf{CF}_{<:}$ subcapturing.

3.1.3 Operational Semantics

Reduction in $\mathsf{CF}_{<:}$ can be stated almost exactly the same as in call-by-value System $\mathsf{F}_{<:}$. Figure 3.2 defines the operational semantics with a single congruence rule that takes an evaluation context E. The only difference compared to System $\mathsf{F}_{<:}$ is that reducing term applications with (BETA-V) needs to additionally replace capture occurences of the lambda parameter. The parameter's term and capture occurences need to be substituted differently; the term occurences are replaced by the value applied to the lambda and the capture occurences are replaced by what the value captured: its *free variables.* (This is in contrast to SCC where the same substitution applies to both term and capture occurences, since we always replace one variable with another thanks to MNF.)

3.1.4 Subcapturing

$\mathsf{CF}_{<:}$ subcapturing is exactly the same as in SCC. The presented rules are equivalent to the ones presented in the ArXiv version [Boruch-Gruszecki et al. 2021], although they are formulated slightly differently. In particular we do not state a separate rule for {**cap**}, since all well-formed $\mathsf{CF}_{<:}$ capture sets anyway subcapture {**cap**} just like they did in SCC.

3.1.5 Subtyping

Due to the type/pretype split, there are technically two subtyping judgements, as shown in Figure 3.4; one for types with rules (CAPT) and (TVAR) and one for pretypes with rules (FUN),

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

REFL-TYP
$$\Gamma \vdash T <: T$$

REFL-PRE
$$\Gamma \vdash U <: U$$

TRANS-TYP
$$\frac{\Gamma \vdash R <: S \qquad \Gamma \vdash S <: T}{\Gamma \vdash R <: T}$$

TRANS-PRE
$$\frac{\Gamma \vdash U <: V \qquad \Gamma \vdash V <: W}{\Gamma \vdash U <: W}$$

TOP
$$\Gamma \vdash U <: \top$$

TVAR
$$\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T}$$

CAPT
$$\frac{\Gamma \vdash C_1 <: C_2 \qquad \Gamma \vdash U_1 <: U_2}{\Gamma \vdash U_1 \,\hat{}\, C_1 <: U_2 \,\hat{}\, C_2}$$

FUN
$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma,\, x : S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x : S_1)\, T_1 <: \forall(x : S_2)\, T_2}$$

TFUN
$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma,\, X <: S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall[X <: S_1]\, T_1 <: \forall[X <: S_2]\, T_2}$$

Figure 3.4: $\mathsf{CF}_{<:}$ subtyping of types and pretypes.

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

VAR-CONCRETE
$$\frac{x : U \,\hat{}\, C \in \Gamma}{\Gamma \vdash x : U \,\hat{}\, \{x\}}$$

VAR-TVAR
$$\frac{x : X \in \Gamma}{\Gamma \vdash x : X}$$

SUB
$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T <: S}{\Gamma \vdash t : S}$$

ABS
$$\frac{\Gamma,\, x : S \vdash t : T \qquad \Gamma \vdash \forall(x : S)\, T \ \textbf{wf}}{\Gamma \vdash \lambda(x : S)\, t \;:\; (\forall(x : S)\, T) \,\hat{}\, (\mathrm{fv}(t) \setminus x)}$$

APP
$$\frac{\Gamma \vdash t : \forall(x : S)\, T \,\hat{}\, C \qquad \Gamma \vdash s : S}{\Gamma \vdash t\, s : [x := \mathrm{cs}(S)]\, T}$$

T-ABS
$$\frac{\Gamma,\, X <: S \vdash t : T \qquad \Gamma \vdash \forall[X <: S]\, T \ \textbf{wf}}{\Gamma \vdash \Lambda[X <: S]\, t \;:\; (\forall[X <: S]\, T) \,\hat{}\, (\mathrm{fv}(t) \setminus x)}$$

T-APP
$$\frac{\Gamma \vdash t : (\forall[X <: S]\, T) \,\hat{}\, C \qquad \textbf{cap} \notin \mathrm{cv}(S)}{\Gamma \vdash t\,[S] : [X := S]\, T}$$

Figure 3.5: $\mathsf{CF}_{<:}$ typing.

(TFUN), and (TOP). Reflexivity and transitivity apply to each kind of judgement; they are the only duplicated rules. The subtyping rules are a straightforward extension of the subtyping rules for System $\mathsf{F}_{<:}$; the only significant departure is the addition of (CAPT) for reasoning with capture sets in types.

3.1.6 Typing

Figure 3.5 shows the typing rules for $\mathsf{CF}_{<:}$. Compared to SCC, there are two new typing rules for type abstractions and a new typing rule for variables bound at a type which itself is a variable. The differences between $\mathsf{CF}_{<:}$ and System $\mathsf{F}_{<:}$ are analogous to the differences between SCC and $\lambda_{<:}$, i.e., adjustments are minimal and only made to account for capture sets and dependent types.

There are two typing rules for variables. Rule (VAR-CONCRETE) types variables x bound at a concrete type $x : U \wedge C \in \Gamma$, and rule (VAR-TVAR) types variables x bound at a type which is a variable $x : X \in \Gamma$. The first rule types the variable as capturing $\{x\}$, which allows assigning precise types to lambda terms which return their arguments (see the choice term in Section 2.2.4). On the other hand, the second typing rule has no capture set to refine: a type variable already stands for a full type. Hence, it types variables simply as X.

Rule (T-ABS) types type abstraction forms and is exactly analogous to rule (ABS), which types term abstractions. Rule (T-APP) types type application forms $t\,[T]$ and is more interesting it prevents instantiating type variables with types capturing **cap** via the $\mathbf{cap} \notin \mathrm{cs}(T)$ premise. As I explained in Section 2.1.5, this restriction helps when extending the core system with concrete capabilities. I will come back to this point later in Section 3.3.3, when we present an extension of the core system with a primitive scoped capability.

Well-formedness constraints

Both (ABS) and (T-ABS) explicitly require the assigned types to be well-formed. I motivate this in the following section.

3.1.7 Well-Formedness

In System $\mathsf{F}_{<:}$, a type is well-formed simply if all type variables mentioned in it are bound in the environment. The $\mathsf{CF}_{<:}$ well-formednesss rules (Figure 3.6) are more complicated: they also track the variance at which term variables appear in capture sets embedded within a type. Put briefly, the rules disallow term variables bound in the type from occuring in contravariant positions.

Doing so is needed because, due to $\mathsf{CF}_{<:}$ terms not being in MNF, there is a significant difference between the static semantics (i.e., typing) and the dynamic semantics (i.e., reduction). Consider a type abstraction $\lambda(x : S')\, t : \forall(x : S)\, T$ (where we allow for some subtyping slack $S <: S'$) and an argument $v : S$. The (APP) typing rule types an application of the lambda to the argument by replacing occurences of x in T with $\mathrm{cs}(S)$, i.e., the capture set of the *type*. When typing term application with (APP), the lambda's parameter x is replaced with the cs of the argument's type S. Meanwhile, the (BETA-V) reduction rule instead substitutes occurences of x in t with $\mathrm{fv}(v)$, and in general we only have $\mathrm{fv}(v) <: \mathrm{cs}(S)$! The static capture set $\mathrm{cs}(S)$ only approximately predicts the free variables $\mathrm{fv}(v)$.

There are two ways to think about this fact. One is that the capture set of the argument's type can be widened through subtyping and subcapturing; another is that the capture set of the argument's type is term-dependent, and hence can shrink under reduction. To illustrate this, let us consider the term:

$$\mathsf{f} = \lambda\,(x : U \wedge \{\mathbf{cap}\})\ \lambda\big(y : U \wedge \{x\}\big)\ y$$

Well-formedness $\boxed{\Gamma\,;\,A_+\,;\,A_- \vdash T\ \mathbf{wf}}$

$$\textsc{capt-wf}\quad \frac{\forall x \in C.\, x \in A_+ \qquad C \subseteq \mathrm{dom}(\Gamma) \qquad \Gamma\,;\,A_+\,;\,A_- \vdash U\ \mathbf{wf}}{\Gamma\,;\,A_+\,;\,A_- \vdash U\,\hat{}\,C\ \mathbf{wf}}$$

$$\textsc{root-wf}\quad \frac{\Gamma\,;\,A_+\,;\,A_- \vdash U\ \mathbf{wf}}{\Gamma\,;\,A_+\,;\,A_- \vdash U\,\hat{}\,\{\mathbf{cap}\}\ \mathbf{wf}} \qquad\qquad \textsc{tvar-wf}\quad \frac{X <: T \in \Gamma}{\Gamma\,;\,A_+\,;\,A_- \vdash X\ \mathbf{wf}}$$

$$\textsc{fun-wf}\quad \frac{\Gamma\,;\,A_-\,;\,A_+ \vdash S\ \mathbf{wf} \qquad \Gamma,\, x : S\,;\,A_+ \cup \{x\}\,;\,A_- \vdash T\ \mathbf{wf}}{\Gamma\,;\,A_+\,;\,A_- \vdash \lambda\,[x : S]\ T\ \mathbf{wf}}$$

$$\textsc{tfun-wf}\quad \frac{\Gamma\,;\,A_-\,;\,A_+ \vdash S\ \mathbf{wf} \qquad \Gamma,\, X <: S\,;\,A_+\,;\,A_- \vdash T\ \mathbf{wf}}{\Gamma\,;\,A_+\,;\,A_- \vdash \forall\,[X <: S] \to T\ \mathbf{wf}} \qquad\qquad \textsc{top-wf}\quad \Gamma\,;\,A_+\,;\,A_- \vdash \top\ \mathbf{wf}$$

Figure 3.6: $\mathsf{CF}_{<:}$ type well-formedness.

applied to a pure value v of type $U\,\hat{}\,\{\}$. Notice that x occurs contravariantly in the capture set of parameter y. By (BETA-V), f v reduces to $\lambda\big(y : U\,\hat{}\,\{\}\big)\ y$, with type $(\forall(y : U\,\hat{}\,\{\})\,U\,\hat{}\,\{y\})\,\hat{}\,\{\}$. However, by applying the subtyping rule (CAPT), we may also assign v the type $U\,\hat{}\,\{\mathbf{cap}\}$, and hence type the application $\mathsf{f}(v)$ with the type $(\forall(y : U\,\hat{}\,\{\mathbf{cap}\})\,U\,\hat{}\,\{y\})\,\hat{}\,\{\}$. This is unsound, as the function type $(\forall(y : U\,\hat{}\,\{\})\,U\,\hat{}\,\{y\})\,\hat{}\,\{\}$ is categorically *not* a subtype of $(\forall(y : U\,\hat{}\,\{\mathbf{cap}\})\,U\,\hat{}\,\{y\})\,\hat{}\,\{\}$; it can be applied to strictly fewer values.

The discrepancy between the typing rules and the operational semantics motivates the $\mathsf{CF}_{<:}$ well-formedness judgement, defined with a triple context $\Gamma\,;\,A_+\,;\,A_- \vdash T\ \mathbf{wf}$. Here, Γ is the standard environment and A_+ and A_- are sets of term variables. A term variable x can appear covariantly only if it occurs in A_+, and contravariantly only if it occurs in A_-. For brevity, when dealing with sets that contain both term and type variables, I write $\Gamma\,;\,A_+\,;\,A_- \vdash T\ \mathbf{wf}$ instead of $\Gamma\,;\,A_+ \cap D\,;\,A_- \cap D \vdash T\ \mathbf{wf}$ where D is the set of term variables bound in Γ. I also define $\Gamma \vdash T\ \mathbf{wf}$ as $\Gamma\,;\,\mathrm{dom}(\Gamma)\,;\,\mathrm{dom}(\Gamma) \vdash T\ \mathbf{wf}$. To ensure that subtyping holds with respect to term-dependent capture sets, rules (CAPT-WF), (FUN-WF) and (TFUN-WF) together ensure that a term variable x in a type T can only occur in covariant position with respect to its binding form. This notion is formalized in Section 3.2.

As we have seen, the well-formedness condition prevents direct coupling of capture sets at different polarities. This is less of a restriction than it might seem, since we can express the same coupling through a type variable. The following version of function f does typecheck.

$$\mathsf{f}' = \Lambda[X <: U\,\hat{}\,\{\mathbf{cap}\}]\,\lambda(x : X)\,\lambda(y : X)\,y$$

Note also that the well-formedness restriction only applies to the variable occurrences *in the type* of the lambda; it does not apply when the variable occurs in the actual body of the lambda. For a concrete example, the function

$$\mathrm{g} = \lambda(x : U \,\hat{}\, \{\mathbf{cap}\})\,(\lambda(y : U \,\hat{}\, \{x\})\, y)\, x$$

is well-typed with type $\forall(x : U \,\hat{}\, \{\mathbf{cap}\})\, U \,\hat{}\, \{x\}$, even though x is captured at negative polarity in the second lambda.

3.2 Metatheory

I proceed to presenting the metatheoretic properties of $\mathsf{CF}_{<:}$. The thesis is accompanied by a full mechanization using the Coq theorem prover, described in more detail in Section 3.2.1. First, observe that $\mathsf{CF}_{<:}$ is indeed a straightforward extension of System $\mathsf{F}_{<:}$. In particular, erasing capture sets from well-typed $\mathsf{CF}_{<:}$ terms yields well-typed System $\mathsf{F}_{<:}$ terms.

Lemma 3.1 (Erasure)**.** *Let t be a $\mathsf{CF}_{<:}$ term such that $\vdash\ t : T$ for some type T. Let $\lceil\cdot\rceil$ be a function from $\mathsf{CF}_{<:}$ terms and types to System $\mathsf{F}_{<:}$ terms and types that erases capture sets (and thereby all term dependencies). Then we have $\vdash : \lceil t\rceil : \lceil T\rceil$.*

Proof. Immediate from structural induction on the typing derivation of $\vdash\ t : T$. □

Moreover, System $\mathsf{F}_{<:}$ embeds naturally into $\mathsf{CF}_{<:}$, simply by annotating System $\mathsf{F}_{<:}$ function and type abstraction types with either the empty or the universal capture set.

Lemma 3.2 (Embedding)**.** *Let t be a System $\mathsf{F}_{<:}$ term such that $\vdash\ t : T$ for some type T. Let $\lfloor\cdot\rfloor$ be a function from System $\mathsf{F}_{<:}$ to $\mathsf{CF}_{<:}$ terms and types that annotates System $\mathsf{F}_{<:}$ types of function and type abstractions with $\{\}$. Then we have that $\vdash\ \lfloor t\rfloor : \lfloor T\rfloor$.*

Proof. Structural induction on the typing derivation, after observing that every term variable will be a subcapture of $\{\}$. □

All of the following lemmas and theorems are mechanized in Coq.

Soundness

$\mathsf{CF}_{<:}$ satisfies the standard Progress and Preservation lemmas.

Theorem 3.1 (Progress)**.** *If $\vdash\ t : T$, then either t is a value, or there exists a term t' such that we can take a step $t \longrightarrow t'$.*

Theorem 3.2 (Preservation)**.** *If $\Gamma\ \vdash\ t_1 : T$ and $t_1 \longrightarrow t_2$, then we have that $\Gamma\ \vdash\ t_2 : T$.*

Substitution Lemmas

Due to term-dependency of $\mathsf{CF}_{<:}$, a few nonstandard substitution lemmas are necessary to prove the Progress and Preservation theorems. Once again, this is due to the difference between the static and dynamic semantics. This difference necessitates the following lemma, linking the static and dynamic capture sets.

Lemma 3.3 (Term Substitution Preserves Typing). *If* $\Gamma,\ x : S \vdash t : T$ *and* $\Gamma,\ x : S\ ;\ \{x\} \cup \mathrm{dom}(\Gamma)\ ;\ \mathrm{dom}(\Gamma) \vdash T$ ***wf****, then for all* v *such that* $\Gamma \vdash v : T$*, we have*

$$\Gamma \vdash [x := v, x := \mathrm{fv}(v)]\, t : [x := \mathrm{cs}(S)]\, T.$$

Without the well-formedness condition, we would only be able to show that

$$\Gamma \vdash [x := v, x := \mathrm{fv}(v)]\, t : [x := \mathrm{fv}(v)]\, T.$$

Now, as $\Gamma \vdash \mathrm{fv}(v) <: \mathrm{cs}(S)$, and as x does not occur contravariantly in T due to our well-formedness constraints, we have that $\Gamma \vdash [x := \mathrm{fv}(v)]\, T <: [x := \mathrm{cs}(S)]\, T$. Formally, this is stated below in the following lemma, which is needed to prove Lemma 3.3:

Lemma 3.4 (Monotonicity of Covariant Capture Set Substitution). *If* $\Gamma, x : S\ ;\ \mathrm{dom}(\Gamma) \cup \{x\}\ ;\ \mathrm{dom}(\Gamma) \vdash T$ ***wf****, then for all* C_1, C_2 *such that* $\Gamma \vdash C_1 <: C_2$*, we have:*

$$\Gamma \vdash [x := C_1]\, T <: [x := C_2]\, T$$

Meaning of capture sets

Observe that the capture of a value's type accounts for the value's free variables.

Lemma 3.5 (Capture Prediction for Values). *If* $\Gamma \vdash v : T$*, then* $\Gamma \vdash \mathrm{fv}(v) <: \mathrm{cs}(T)$.

Proof. Induction on the typing derivation $\Gamma \vdash v : T$. Now, as v is a value, the base case is either an application of the typing rule (ABS) or (T-ABS), and hence $T = U \mathbin{\hat{}} \mathrm{fv}(v)$ for some pretype U, as desired. Inductively, we have an application of the typing rule (SUB). Hence $\Gamma \vdash v : T'$, $T' <: T$, and $\Gamma \vdash \mathrm{fv}(v) <: \mathrm{cs}(T')$. Now, as v is a value, $T' = U' \mathbin{\hat{}} C'$ for some capture set C' and pretype U', and hence $T = U \mathbin{\hat{}} C$ for some capture set C and pretype U. Hence $\mathrm{fv}(v) <: \mathrm{cs}(T') = C' <: C = \mathrm{cs}(T)$, as desired. $\square$

Note that $\mathrm{fv}(v)$ and $\mathrm{cs}(T)$ are related via *subcapturing*: the former is not necessarily a subset of the latter. For example, consider a value $v = \lambda(x : \top \mathbin{\hat{}} \{\mathbf{cap}\}).\ y$ in a typing environment $\Gamma = (x : \top \mathbin{\hat{}} \{\})$. Here we may assign v the type $T = (\top \mathbin{\hat{}} \{\mathbf{cap}\} \to \top) \mathbin{\hat{}} \{\}$ by subsuming away the capture set for x, but we also have that $\Gamma \vdash (\mathrm{fv}(v) = \{x\}) <: (\{\} = \mathrm{cs}(T))$.

The following theorem captures the essence of capture tracking in $\mathsf{CF}_{<:}$. From the Preservation Theorem and the Capture Prediction for Values Lemma, it follows that $\mathsf{CF}_{<:}$ capture sets accurately track the captured variables of the value a term reduces to.

Theorem 3.3 (Capture Prediction for Terms)**.** *If* $\Gamma \vdash t : T$ *and* $t \longrightarrow^* v$*, then* $\Gamma \vdash \mathrm{fv}(v) <: \mathrm{cs}(T)$.

Furthermore, observe that Theorem 3.3 can be applied to an arbitrary subterm of a term.

Corollary 3.1 (Capture Prediction in Context)**.** *Let* $C[s]$ *be a term well-typed in the empty environment, where* C *is an arbitrary term context with a single hole. Then* $\Gamma \vdash s : T$ *for some* Γ *and* T*. If* $s \longrightarrow^* v$*, then Theorem 3.3 applies and we have* $\Gamma \vdash \mathrm{fv}(v) <: \mathrm{cs}(T)$.

The theorem has important consequences, despite appearing deceptively simple. In core $\mathsf{CF}_{<:}$, it allows predicting the free variables captured by the value a term reduces to. If we extend $\mathsf{CF}_{<:}$ with capabilities such that certain reduction steps only apply to redexes with a capability, then the same theorem lets us reason about what reduction steps a term might take!

As a brief illustration of the idea, consider extending $\mathsf{CF}_{<:}$ with the classical store-and-location representation of mutable ML references [Pierce 2002]. In such a setting, free variables may be replaced with locations. Assuming that locations are typed as capabilities, one application of Theorem 3.3 is determining if a locally-created mutable reference can leak to the outer context surrounding the term, i.e., if it is local to a particular subterm or not.

We will come back to this idea when discussing the extensions to $\mathsf{CF}_{<:}$ in Section 3.3.2, Section 3.3.3, Section 3.3.4 and Section 3.3.5: we will see that Theorem 3.3 is the formal basis ensuring the soundness of each extension.

3.2.1 Mechanization

The $\mathsf{CF}_{<:}$ proof of soundness was mechanized using the Coq theorem prover [Coq 2004; Bertot and Castéran 2004]. The mechanization includes the soundness proofs for both the core calculus and the non-local returns extension (Section 3.3.3). As $\mathsf{CF}_{<:}$ is an extension of System $\mathsf{F}_{<:}$, the Coq mechanization was based on the locally nameless proof of System $\mathsf{F}_{<:}$ by Aydemir et al. [2008]. In particular, since $\mathsf{CF}_{<:}$ types can mention term variables, we chose the locally-nameless approach to avoid problems with alpha-equivalence of types. I highlight two interesting aspects of the mechanization.

Mechanizing Capture Sets

Capture sets in $\mathsf{CF}_{<:}$ are mechanized as a product of sets, one per each kind of capture set member. In total, the product had four elements: a boolean representing an occurence of **cap**, a set of bound variables (de Bruijin indices), a set of free variables, and a set of labels (introduced by the extensions). To support handling this construct, we have copied Coq's FSet library and adapted its tactics to work with our capture sets. This worked well for the most part, aside from some issues which are to be expected when usings sets. In particular, we often had sets that were only equal propositionally (and not definitionally), for example $\{1,2,3\}$ and $\{1\} \cup \{2\} \cup \{3\}$. The number of such cases required adjustments to how the proofs are written so that the required proofs of propositional equality could be discharged without too much manual work, something which was not a noticeable problem for us in other contexts where we could rely on Coq's support for definitional equality.

Mechanizing Well-Formedness

The $\mathsf{CF}_{<:}$ well-formedness judgement (Figure 3.6) needs to keep track of two term variables sets, A_+ and A_-, which describe the variables in co- and contravariants position relative to the current location in the type. A previous version of the judgment had A_* that were not be sets, but would actually bind the appropriate variables. However, mechanizing this version required showing that well-formedness is preserved under permutation, and such proofs are (reportedly) known to be challenging in Coq. We instead opted to use sets, which in the end we indeed found more tractable.

3.3 Evaluation

Capture sets in the system presented so far faithfully track free variables of values. The system, however, does not define any concrete capabilities with any particular operational semantics. Capture Tracking can be used to track many different sorts of capabilities, none of which should be privileged by being included in the base system.

To evaluate the base system, we will first see the signatures assigned in $\mathsf{CF}_{<:}$ to common data types, and then we will see how to extend the base system with capabilities and how the metatheory of the base system is used to ensure the soundness of each extension.

3.3.1 Data Structures in $\mathsf{CF}_{<:}$ - List

To give some intuition for the calculus, we first take a look at the type signatures of different versions of the map function, which maps an arbitrary function argument over a list of values. We can encode List in $\mathsf{CF}_{<:}$ using the standard right-fold Böhm-Berarducci encoding [Böhm and Berarducci 1985]. First, recall that $(S \to T)\,\hat{}\,C$ is a shorthand for $(\forall(x : S)\, T)\,\hat{}\,C$. We write $\mathsf{List}[T]$ as an abbreviation for the type

$$\mathsf{List}[T] \;=\; (\forall[X <: \top\,\hat{}\,\{\mathbf{cap}\}]\, \forall(op : \mathsf{Op}[T,X])\, (\forall(z : X)\, X)\,\hat{}\,\{op\})\,\hat{}\,\mathrm{cs}(T),$$

where $\mathsf{Op}[T,X]$ is in turn an abbreviation for:

$$\mathsf{Op}[T,X] \;=\; (T \to (X \to X)\,\hat{}\,\{\mathbf{cap}\})\,\hat{}\,\{\mathbf{cap}\}.$$

A list is encoded as a folding function which takes (1) a default value z of some desired result type C to be returned when the list is empty and (2) a folding operation op to be applied when the list is non-empty. Notice that the capture set of the list is $\mathrm{cs}(T)$: the list type captures whatever the list's elements capture.

We can type a strict map function strictMap, which applies some function f to each element of the list, as follows:

$$\begin{aligned}\mathsf{strictMap}:\ & \forall[A <: \top\,\hat{}\,\{\mathbf{cap}\}]\\ & \forall[B <: \top\,\hat{}\,\{\mathbf{cap}\}]\\ & \forall(xs : \mathsf{List}[A])\\ & (\forall(f : (A \to B)\,\hat{}\,\{\mathbf{cap}\})\\ & \mathsf{List}[B])\,\hat{}\,\{xs\}\end{aligned}$$

The function argument f to map may capture arbitrary capabilities. As map is strict, that capability is not retained in the final result type. It is, however, retained by the last curried closure, since to produce the output of the map we naturally need the input list.

We can also consider a version of map which requires its function argument f to be pure. Here is its signature.

$$\begin{aligned}\mathsf{pureMap}:\ & \forall[A <: \top\,\hat{}\,\{\mathbf{cap}\}]\\ & \forall[B <: \top\,\hat{}\,\{\mathbf{cap}\}]\\ & \forall(xs : \mathsf{List}[A])\\ & \forall(f : A \to B)\\ & \mathsf{List}[B]\end{aligned}$$

This version of map enforces that the mapping function does not access any capabilities, which is desireable if the implementation maps the collection in parallel and we want to ensure that the result of mapping is deterministic.

Lazy lists, which evaluate their elements lazily only when the elements are accessed, can be modelled as lists of thunks $\mathsf{List}[(\mathsf{Unit} \to A)\,\hat{}\,\{A\}]$. Then a lazyMap function, which creates a list of thunks that apply a given function f to the result of forcing each thunk of a given input list, can be typed as follows.

$$\begin{aligned}\mathsf{lazyMap}:\ & \forall[A <: \top\,\hat{}\,\{\mathbf{cap}\}]\\ & \forall[B <: \top\,\hat{}\,\{\mathbf{cap}\}]\\ & \forall(xs : \mathsf{List}[(\mathsf{Unit} \to A)\,\hat{}\,\{A\}])\\ & (\forall(f : (A \to B)\,\hat{}\,\{\mathbf{cap}\})\\ & \mathsf{List}[(\mathsf{Unit} \to B)\,\hat{}\,\{A, f\}])\,\hat{}\,\{xs\}\end{aligned}$$

Note that the elements of the output list have capture set $\{A, f\}$: they capture the original elements of type A and the function f, but not any values of type B. Values of type B will be computed only when the elements of the output list are forced. If f is pure, we can enforce this in its type; then f is not tracked, so it does not need to be included in the capture set of the elements of the output list.

$$\begin{aligned}\mathsf{lazyPureMap}:\ &\forall[A <: \top\,\hat{}\,\{\mathbf{cap}\}]\\ &\forall[B <: \top\,\hat{}\,\{\mathbf{cap}\}]\\ &\forall(xs : \mathsf{List}[(\mathsf{Unit} \to A)\,\hat{}\,\{A\}])\\ &\forall(f : A \to B)\\ &\mathsf{List}[(\mathsf{Unit} \to B)\,\hat{}\,\{A\}]\end{aligned}$$

3.3.2 Abort

We will now see a number of progressively more complex extensions to $\mathsf{CF}_{<:}$, with each extension adding some form of primitive capabilities to the system. Since the extensions introduce capabilities which affect the operational semantics, it becomes interesting to ask if a given term may access a particular capability. As we will see, the metatheory of the base system can be used to perform such reasoning even after the system is extended.

The first extension (Figure 3.7) is rather simple: it adds a basic capabilitity which allows aborting the entire computation. The $\mathsf{Abort}.\mathbf{do}$ term form, when encountered during reduction in a non-empty context, causes the entire surrounding term to reduce to $\mathsf{Abort}.\mathbf{do}$. Since the $\mathsf{Abort}.\mathbf{do}$ form is not a value, the Progress Theorem is now stated as follows:

Theorem 3.4 (Progress (abort variant))**.** *If* $\vdash t : T$*, then* t *either is a value, or it is the* $\mathsf{Abort}.\boldsymbol{do}$ *term, or there exists a term* t' *such that we can take a step* $t \longrightarrow t'$*.*

Importantly, Lemma 3.5 can be used to prove the following lemma:

Lemma 3.6 (Abort Blame Assignment)**.** *Let* $v\ s$ *be a term well-typed in the empty environment. Then* $\vdash v : (\forall(x : S)\ T)\,\hat{}\,C$ *and* $\vdash s : S$ *for some* C, S *and* T*. If* $\mathsf{Abort} \notin C \cup \mathsf{cs}(S)$*, then* $v\ s$ *aborts (reduces to* $\mathsf{Abort}.\boldsymbol{do}$ *in one or more steps) only if* s *aborts.*

Proof. By induction on steps of the $v\ s \longrightarrow^* \mathsf{Abort}.\mathbf{do}$ relation. Since v is a value and $\mathsf{Abort} \not\in C$, by Lemma 3.5 and inspection of the subcapturing rules we know that $\mathsf{Abort} \not\in \mathrm{fv}(v)$. If $v\ s \longrightarrow \mathsf{Abort}.\mathbf{do}$, then $v\ s$ must be of the form $\mathsf{E}[\mathsf{E}'[\mathsf{Abort}.\mathbf{do}]]$, where $s = \mathsf{E}'[\mathsf{Abort}.\mathbf{do}]$, which means that s aborts as well. Otherwise, if $v\ s$ steps with (BETA-V), then s is a value. Again by Lemma 3.5 we then know that $\mathsf{Abort} \notin \mathrm{fv}(v\ s)$, which means that we cannot have $v\ s \longrightarrow^* \mathsf{Abort}.\mathbf{do}$ – a contradiction. Otherwise, $v\ s$ can step to $v\ s'$ such that $s \longrightarrow s'$, in which case we can conclude by Theorem 3.2 and IH. □

This lemma demonstrates that the capture sets assigned by typing to terms can be used to reason about what subterm can cause the entire expression to abort. The type of v tells us that it is *abort-safe,* i.e., it will not cause the evaluation to abort no matter whether its argument s mentions abort or not. In terms of a complete programming language, a term like $v\ s$ corresponds to a function call. Thus, the above lemma tells us that if the execution of such

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

$$
\begin{array}{rcll}
q & ::= & \ldots \mid & \mathsf{Abort} \\
t & ::= & \ldots \mid & \mathsf{Abort}.\mathbf{do}
\end{array}
$$

ABORT
$$\Gamma \vdash \mathsf{Abort}.\mathbf{do} : T$$

Reduction $\boxed{t \longrightarrow t}$

BETA-ABORT
$$\frac{\mathsf{E} \neq [\,]}{\mathsf{E}[\mathsf{Abort}.\mathbf{do}] \longrightarrow \mathsf{Abort}.\mathbf{do}}$$

Figure 3.7: Typing and operational semantics $\mathsf{CF}_{<:}$ extended with aborting.

a call aborts, based on the type of the function v we know we can assign blame for aborting to the argument s.

3.3.3 Non-Local Returns

The next extension illustrate how $\mathsf{CF}_{<:}$ can be applied to the problem of scoped effects, i.e., effects which can only be performed in a limited scope, typically because after the scope is left the entity which *handles* the effect becomes unavailable. I discuss extensions for non-local returns, regions and effect handles. Exceptions are another commonplace example of a scoped effect, and other examples can be found in the literature on *effect masking* [Biernacki et al. 2017; Mcbride and Wadler 2019; Zhang and Myers 2019].

The basis for all the extensions is the same in every case. Following the object capability model, the effectful operation is guarded via a capability object introduced in the scope in which the effect is available. Capture Tracking can guarantee that this capability is *not accessed* after the scope is left, ensuring that all effect operations are well-scoped.

The first extension we inspect is a simple control effect: a non-local return. A non-local return allows transferring the control flow to the end of a particular block, without necessarily being within the lexical scope of that block. In contrast to a more classical return statement, non-local returns work across method and function boundaries. Syntax and typing rules of the extension are defined in Figure 3.8.

Example

We now take a look at a small term that demonstrates non-local returns. The term represents a program which sums up the square roots of a list of numbers, returning NaN if one of the

Syntax

$U ::= \ldots$	**Pretypes**	$t ::= \ldots$	**Terms**
$\mathsf{Return}[T]$	return capability	$\textbf{handle}\, x : \mathsf{Return}[T]\, \textbf{in}\, t$	return-able block
		$t.\textbf{return}\, s$	explicit return

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

RETURN
$$\frac{\Gamma,\ x : \mathsf{Return}[T]\,\hat{}\,\{\mathbf{cap}\} \vdash t : T \qquad \mathbf{cap} \notin \mathrm{cs}(T)}{\Gamma \vdash \textbf{handle}\, x : \mathsf{Return}[T]\, \textbf{in}\, t : T}$$

DO-RETURN
$$\frac{\Gamma \vdash t : \mathsf{Return}[S]\,\hat{}\,C \qquad \Gamma \vdash s : S}{\Gamma \vdash t.\textbf{return}\, s : T}$$

Figure 3.8: Static semantics of $\mathsf{CF}_{<:}$ extended with non-local returns.

numbers is negative.

```
root =
    λ(x : Double) λ(ret : (∀(d : Double) Double)^{cap})
        if x < 0 then ret NaN else sqrt x

sumRoots =
    λ(xs : List[Double]) λ(ret : (∀(d : Double) Double)^{cap})
        xs (λ(x : Double) λ(y : Double) y + (root x ret)) 0.0

handle r : Return[Double] in
    sumRoots [1.0, 2.0, 3.0, −1.0] (λ(x : Double) r.return x)
```

The program is partitioned into three parts. First, the root function calculates the square root of its first argument. In case the argument is negative, it uses its second argument (intended to be a return capability), calling it with NaN as an argument. Second, the sumRoots function applies root to each element of a list and sums up the results, using the right-fold encoding of List (Section 3.3.1). It directly passes the *ret* function to root. Finally, the "main body" of the program first introduces the r capability and then calculates the sum of the elements of a particular list, using r to allow sumRoots to return early if an inappropriate list element is found.

Note how the dynamic call to *ret* in function root is not in the lexical scope of the handler introducing r: the non-local return capability allows root to return from a lexical scope in its calling context, unlike a more typical **return** statement.

The program is well-typed, since the r capability is not captured by the result of application of sumRoots. On the other hand, the following simple variation is ill-typed.

$$\begin{array}{l}\textbf{handle}\, \mathsf{r} : \mathsf{Return}[\mathsf{Double}]\ \textbf{in} \\ \quad \lambda()\, \mathsf{sumRoots}\, [1.0, 2.0, 3.0, -1.0]\ (\lambda(x : \mathsf{Double})\ \mathsf{r}.\textbf{return}\, \lambda()\, x)\end{array}$$

Here, by rule (ABS), r does appear in the capture set of the handler's body, which violates the requirement for (RETURN). A version using lazy list (Section 3.3.1) would also be ill-typed.

Aside. Observe that the root and sumRoots are defined in such a way that the *ret* function *might* abort the computation, or it might return a default value instead: since neither function needs to receive direct access to a non-local return capability, it is their caller who decides how the computation is to be carried out. Another reasonable version of *ret* could simply return 0.0, which would calculate the sum of non-negative elements of the list.

Operational semantics

The operational semantics of the non-local returns extension (Figure 3.9) are built around the concept of *labels* l [Biernacki et al. 2020]. Labels are special forms, a formal representation of runtime capabilities. They only exist when reducing terms: rule (BETA-RETURN-INTRO) generates a fresh label when a non-local return scope is entered. Conceptually, labels represent the *location* of an element of the stack, used to precisely look up which scope a non-local return should terminate. In multiple senses, a label is like a variable: it needs to be bound to a type in an environment (the *signature environment*), it can be either free or bound in a term, and it is counted as a variable by fv (Figure 3.9). There are three key differences between labels and variables. Labels are values (allowing them to be directly passed as arguments), labels are never substituted, and types are well-formed even if they contain unbound labels. The latter is permissible, since subcapturing treats labels solely based on their identity, i.e., they are never looked up in the environment.

The operational semantics introduce three new reduction rules and three new evaluation contexts. The most important new evaluation context form is **handle** $l : T$ **in** E: it allows reducing the term *underneath* the **handle** binder. There are two ways for the binder can be removed during reduction. First, by reducing to a value and applying rule (BETA-RETURN), which corresponds to normally returning from a block. Second, the term inside the block can invoke the return capability and explicitly return from it, which corresponds to the (CONTEXT-RETURN) reduction rule. Note that a term that tries to invoke a return capability after its binder was removed would be stuck.

Syntax and definitions

$l ::= \texttt{@a13}, \texttt{@4f1}, \ldots$	runtime labels
$v ::= \ldots \mid l$	label value
$c ::= \ldots \mid l$	label capture
$t ::= \ldots \mid \textbf{handle}\, l : \mathsf{Return}[T]\ \textbf{in}\ t$	return scope
$\Sigma ::= \emptyset \mid \Sigma,\ l : \mathsf{Return}[T]$	signature environment

$$\mathrm{fv}(l) = \{l\} \qquad \mathrm{fv}(\textbf{handle}\, l : T\ \textbf{in}\ t) = \mathrm{fv}(t) \setminus l$$

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

SC-LABEL

$$\Gamma \vdash \{l\} <: \{\mathbf{cap}\}$$

Type assignment $\boxed{\Gamma \mid \Sigma \vdash t : T}$

RETURN-L

$$\frac{l : \mathsf{Return}[T] \in \Sigma \qquad \Gamma \mid \Sigma,\ l : \mathsf{Return}[T] \vdash t : T \qquad \mathbf{cap} \notin \mathrm{cs}(T)}{\Gamma \mid \Sigma \vdash \textbf{handle}\, l : \mathsf{Return}[T]\ \textbf{in}\ t : T}$$

LABEL

$$\frac{l : \mathsf{Return}[T] \in \Sigma}{\Gamma \mid \Sigma \vdash l : \mathsf{Return}[T]\,\hat{}\,\{l\}}$$

Reduction $\boxed{t \longrightarrow t}$

BETA-RETURN

$$\textbf{handle}\, l : \mathsf{Return}[T]\ \textbf{in}\ v \longrightarrow v$$

CONTEXT-RETURN

$$\textbf{handle}\, l : \mathsf{Return}[T]\ \textbf{in}\ \mathsf{E}[l.\textbf{return}\ v] \longrightarrow v$$

BETA-RETURN-INTRO

$$\frac{l\ \textbf{fresh}}{\textbf{handle}\, x : \mathsf{Return}[T]\ \textbf{in}\ t \longrightarrow \textbf{handle}\, l : \mathsf{Return}[T]\ \textbf{in}\ [x := l]\, t}$$

$$\mathsf{E} ::= \ldots \mid \textbf{handle}\, l : T\ \textbf{in}\ \mathsf{E} \mid \mathsf{E}.\textbf{return}\ t \mid l.\textbf{return}\ \mathsf{E}$$

Figure 3.9: Operational semantics of $\mathsf{CF}_{<:}$ extended with non-local returns. The highlighted parts of the judgments are merely a proof device to ensure the soundness of the extension.

Soundness

In this extension it is now possible for a term to be *ill-scoped*, i.e., attempt to access a non-local return capability after the appropriate scope was already removed from the reduction context. This was exemplified by the previously-presented examples, and is also illustrated by the following term.

$$(\textbf{handle}\, r : \mathsf{Return}[\mathsf{Int}]\, \textbf{in}\, \lambda(x : \mathsf{Int})\, r.\textbf{return}\, x)\, 0$$

To statically disallow such terms, we need to ensure that non-local return capabilities are only accessed during their binders dynamic extent, i.e., while the appropriate **handle** form is in the evaluation context.

There are two ways a capability could be accessed after its scope was left: either by being captured by a value returned from the scope the normal way (with rule (BETA-RETURN)), or by being captured by a value returned from the scope via the return capability itself (with rule (CONTEXT-RETURN)). The restriction on $\mathrm{cs}(T)$ in (RETURN) ensures that neither case can occur. Informally, If returning a value v of type T could leak the label l, then $l \in \mathrm{fv}(v)$. Then by Lemma 3.5 and by inspecting the subcapturing rules, it follows that $\{l\} <: \mathrm{cs}(T)$. There are two ways to derive this subcapturing relation: either x or **cap** must be a member of $\mathrm{cs}(T)$. The first is prevented because we implicitly make the assumption that T is well-formed in Γ, the second is explicitly prevent with the mentioned premise of (RETURN). Additionally, observe that preventing type variables from ranging over types capturing cap with the premise of (T-APP) is necessary for the soudness of this extension. Without that premise of (T-APP), we would have trouble, like in the following term (which uses the standard desugaring of let forms).

$$\begin{aligned}
&\textbf{let}\, f = \Lambda[X <: \top \,\hat{}\, \{\textbf{cap}\}]\, \textbf{handle}\, r : \mathsf{Return}[X]\, \textbf{in}\, \lambda(x : X)\, r.\textbf{return}\, x \\
&\textbf{in}\, \textbf{let}\, g = f\, [(\forall(u : \mathsf{Unit})\, \mathsf{Unit}) \,\hat{}\, \{\textbf{cap}\}] \\
&\textbf{in}\, g\, (\lambda(u : \mathsf{Unit})\, ())
\end{aligned}$$

The abstraction f leaks a non-local return capability within a closure. Statically, the result of the non-local return scope is typed with the type variable X, so the premise of (RETURN) is satisfied since $r \notin (\mathrm{cs}(X) = \{X\})$. The rest of the term instantiates X to $(\forall(u : \mathsf{Unit})\, \mathsf{Unit}) \,\hat{}\, \{\textbf{cap}\}$, binds the returned closure to g and proceeds to use it, which will get the term stuck during reduction. Fortunately, the term is ill-typed due to the premise of (T-APP).

3.3.4 Regions

Next, we look at the applicability of $\mathsf{CF}_{<:}$ to region-based memory management [Tofte and Talpin 1997]. Briefly, the idea is to allocate some data structures in *regions*, which in this extension are delimited by a lexical scope. After the lexical scope is left, the entire region can be deallocated in a single operation, which can be more efficient than garbage-collecting or individually deallocating the data structures. Capture Tracking can be used to ensure that after a region is left, the data allocated in the region cannot be accessed anymore, i.e., to statically prevent use-after-free errors for region-allocated data.

Figure 3.10 shows the extensions to $\mathsf{CF}_{<:}$. There are two new major term forms: *regions* are introduced with the $\mathbf{reg}\,x\,\mathbf{in}\;t$ form, and *references* are created with the $x.\mathbf{ref}\;t$ form. The extension also features standard assignment, dereference, and unit term forms. Rule (REGION) types region scope forms. It includes a precondition which ensures that a region reference does not leave its binding scope. Rule (REF) types reference creation forms. To ensure that references allocated on a region cannot be accessed outside out-of-scope, the rule types the newly created references with a capture set accounting for the region. Additionally, to ensure that region references cannot be leaked by being assigned to references in outer regions, a reference's contents type is disallowed from capturing **cap**.

Note that the extension allows creating references on regions which are not statically known: if x and y are regions and we have $s \triangleq \mathbf{if}\;s'\;\mathbf{then}\;x\;\mathbf{else}\;y$, then the type of s would be $\mathsf{Region}\,\hat{}\,\{x, y\}$ and also the following term is well-typed.

$$(\mathbf{if}\;s'\;\mathbf{then}\;x\;\mathbf{else}\;y).\mathbf{ref}\;t : \mathsf{Ref}[T]\,\hat{}\,\{x, y\}$$

Importantly, type-dependent capture polymorphism suffices in many cases where region polymorphism would be otherwise necessary.. For example, the following lambda can dereference a reference allocated on an arbitrary region by typing it as capturing **cap**, whereas region systems would typically require region polymorphism in such cases.

$$\Lambda[X <: \top\,\hat{}\,\{\mathbf{cap}\}]\;\lambda(x : \mathsf{Ref}[X]\,\hat{}\,\{\mathbf{cap}\})\;!x$$

A function can also accept a region reference as an argument in order to allocate a reference on it. The following lambda accepts a region and a reference and duplicates the reference.

$$\Lambda[X <: \top\,\hat{}\,\{\mathbf{cap}\}]\;\lambda(y : \mathsf{Region}\,\hat{}\,\{\mathbf{cap}\})\;\lambda(x : \mathsf{Ref}[X]\,\hat{}\,\{y\})\;y.\mathbf{ref}!x$$

Evaluation

I evaluate the extension by comparing it with Cyclone, a dialect of C notable for featuring region-based memory management. Cyclone regions were limited to lexical scopes [Grossman et al. 2002]; later, "dynamic" regions were added [Fluet et al. 2006]. Dynamic (non-lexical) regions are allocated and de-allocated with a special form; to ensure their soundness, Fluet et al. [2006] use linear types. Since regions in our extension could be similarly refined, I only compare them to lexically-scoped Cyclone regions.

Contrary to Cyclone, the presented extension does not support an "outlives" relationship

Syntax

$v ::= \ldots$	**Terms**
$()$	
$t ::= \ldots$	**Terms**
$\textbf{reg}\, x\, \textbf{in}\, t$	region block
$x.\textbf{ref}\ t$	ref creation
$t := t$	assignment
$!t$	dereference

$U ::= \ldots$	**Core types**
Region	region capability
$\mathsf{Ref}[T]$	reference
Unit	Unit type

Type assignment $\boxed{\Gamma \vdash t : T}$

$$\textsc{Region}\quad \frac{\Gamma,\, x : \mathsf{Region}\,\hat{}\,\{\textbf{cap}\} \vdash t : T \qquad \textbf{cap} \notin \mathrm{cs}(T)}{\Gamma \vdash \textbf{reg}\, x\, \textbf{in}\, t : T}
\qquad
\textsc{Ref}\quad \frac{s : \mathsf{Region}\,\hat{}\,C \in \Gamma \qquad \Gamma \vdash t : T \qquad \textbf{cap} \notin \mathrm{cs}(T)}{\Gamma \vdash s.\textbf{ref}\ t : \mathsf{Ref}[T]\,\hat{}\,C}$$

$$\textsc{Assign}\quad \frac{\Gamma \vdash t : \mathsf{Ref}[S] \qquad \Gamma \vdash s : S}{\Gamma \vdash t := s : \mathsf{Unit}}
\qquad
\textsc{Deref}\quad \frac{\Gamma \vdash t : \mathsf{Ref}[T]}{\Gamma \vdash\, !t : T}$$

Figure 3.10: Static semantics of $\mathsf{CF}_{<:}$ extended with regions.

between regions. This relationship, also called sub-regioning, allows passing references to longer-lived regions where shorter-lived ones are expected. For instance, if we know that the r_1 region is outlived by r_2, we should be able to pass $\mathsf{Ref}[T]\,\hat{}\,\{r_2\}$ where we expected $\mathsf{Ref}[T]\,\hat{}\,\{r_1\}$. We would be able to do so if bounds of r_2 permitted deriving that $\{r_2\} <: \{r_1\}$, i.e., if we could put *lower bounds* on term variables (which matches what Cyclone does).

Still, subcapturing suffices in many cases where sub-regioning would be necessary in other systems. For instance, a conditional expression which returns a reference allocated either on r_1 or on r_2 can be typed with a type of the form $\mathsf{Ref}[T]\,\hat{}\,\{r_1, r_2\}$.

The extension as presented supports regions based on the more widely applicable type system of $\mathsf{CF}_{<:}$, while Cyclone features are much more specialized. Cyclone has a separate concept of region variables ρ and region handles **region**(ρ); region-polymorphic definitions must be explicitly qualified. Cyclone tracks the *use* of regions with an effect system, and, to avoid explicit effect polymorphism, defines a bespoke `regions_of` type operator. In contrast, the Capture Tracking approach does not need a separate effect system, supports region polymorphism without introducing region variables, and does not require unnecessarily qualifying every region-polymorphic function with a region variable.

Syntax

$t ::= \ldots$	**Terms**	$U ::= \ldots$	**Pretypes**
$\textbf{handle}\, x : \mathsf{Eff}[S,T] = \lambda\left(y\,k\right)\ s\ \textbf{in}\ t$	handling	$\mathsf{Eff}[S,T]$	effect capability
$x.\textbf{do}\ t$	handler call		

Type assignment $\boxed{\Gamma \vdash t : T}$

$$\textsc{handle}\quad \frac{\begin{array}{c}\textit{(1a)}\ \ \mathbf{cap} \notin \mathrm{cs}(S) \qquad \textit{(1b)}\ \ \mathbf{cap} \notin \mathrm{cs}(T) \\ \textit{(2)}\ \ \Gamma,\ y : S,\ k : (\forall(x : B)\, R)\,\hat{}\,\{\mathbf{cap}\} \vdash s : R \\ \textit{(3)}\ \ \Gamma,\ x : \mathsf{Eff}[S,T]\,\hat{}\,\{\mathbf{cap}\} \vdash t : R\end{array}}{\Gamma \vdash \textbf{handle}\, x : \mathsf{Eff}[A,B] = \lambda\left(y\,k\right)\ s\ \textbf{in}\ t : R}$$

$$\textsc{do}\quad \frac{\begin{array}{c}\Gamma \vdash s : S \\ \Gamma \vdash t : \mathsf{Eff}[S,T]\,\hat{}\,C\end{array}}{\Gamma \vdash t.\textbf{do}\ s : T}$$

Figure 3.11: Static semantics of CF$_{<:}$ extended with effect handlers.

3.3.5 Effect Handlers

As a final case study on effects, I present a generalization of the non-local returns extension to algebraic effects and handlers [Plotkin and Power 2003; Plotkin and Pretnar 2013]. Effect handlers are a powerful program structuring paradigm: they can be viewed as exceptions which allow resuming the interrupted computation, or otherwise as a more structured version of delimited continuations. They allow expressing various complex control-flow patterns and in fact generalize previously presented extensions, as well as Java-like checked exceptions. The extension given here uses effect handlers in capability-passing style [Brachthäuser and Schuster 2017; Zhang and Myers 2019; Brachthäuser et al. 2020a], since it perfectly fits the Capture Tracking framework. To keep the presentation simple, following Zhang and Myers [2019] effect handlers in this extension are limited to a single operation and cannot have return clauses.

Figure 3.11 extends the core CF$_{<:}$ calculus with additional syntax for effect handlers. The extension features a new pretype $\mathsf{Eff}[S,T]$, representing represents effect operations from S to T. The type argument S indicates the type of values passed to an effect operation, and the type argument T indicates the type of values returned by an effect operation. The extension also features two new term forms. First, the term form $\textbf{handle}\, x : \mathsf{Eff}[S,T] = \lambda\left(y\,k\right)\ s\ \textbf{in}\ t$ introduces the handler capability $x : \mathsf{Eff}[S,T]\,\hat{}\,\{\mathbf{cap}\}$ and binds it within the lexical scope of t. The handler implementation $\lambda\left(y\,k\right)\ s$ has two parameters: y will be bound to the argument of type A passed to the effect operation, while k represents the continuation. To avoid having to annotate the type of the continuation, this term form diverges slightly from the term abstraction form, since the type annotation on x precisely specifies the types of y and k. In the examples, I also sometimes use the shorthand $\textbf{handle}\, x = h\ \textbf{in}\ s$. Second, invoking an effect operation with $x.\textbf{do}\ v$ suspends the current computation, passing the argument v to the handler bound to x.

Operational Semantics

The operational semantics for the effect handler extension are very similar to the ones for the non-local return extensions (Figure 3.9). I present the operational semantics for this extension by discussing each reduction rule separately. The full operational semantics are shown later (Figure 3.12). Again, the extension uses runtime labels to mark positions on the stack. Like with non-local returns, those runtime labels are introduced by the following rule.

BETA-HANDLE-INTRO

$$\frac{l \text{ fresh}}{\textbf{handle}\, x = \lambda\left(y\, k\right)\ s\ \textbf{in}\ t \longrightarrow \textbf{handle}\, l = \lambda\left(y\, k\right)\ s\ \textbf{in}\ [x \mapsto l]\,t}$$

Since reducing a handler generates a fresh label, the effect handlers in this extension are *generative* [Biernacki et al. 2020]. Additionally, there are two new reduction rules. The first rule removes the handler binder if the program w is already a value. This is only safe when w does not mention the handler capability; otherwise we could perform an effect which has no handler on the stack. As expected based on previous extensions, the typing rules ensure that this is never the case.

BETA-HANDLE

$$\textbf{handle}\, l = h\ \textbf{in}\ w \longrightarrow w$$

The second rule reduces terms which invoke an effect operation, using the corresponding handler to do so.

CONTEXT-HANDLE

$$\frac{\mathsf{E} = \textbf{handle}\, l = \lambda\left(y\, k\right)\ s\ \textbf{in}\ \mathsf{E}'}{\mathsf{E}[l.\textbf{do}\ v] \longrightarrow [k := \lambda(z)\, \mathsf{E}[z], y := v]s}$$

To reduce a call to an effect operation $l.\textbf{do}\ v$ in a context E, the context must provide a handler bound at label l. Furthermore, the evaluation context between the handler and the effect operation call is denoted by E'. The entire term is reduced by taking the handler body s and substituting the argument v for y and the continuation $\lambda(z)\,\mathsf{E}[z]$ for k into it. Calling the continuation will reinstantate the delimited evaluation context E; this context also contains the handler bound at l. The presented operational semantics thus implement *deep handlers* [Kammar et al. 2013].

Typing

The typing rules in Figure 3.11 are naturally more complex than the ones for non-local returns, but the core principle stays the same: in both cases the (HANDLE) rule ensures that the scoped handler capability x does not outlive the scope that binds it. The premises of (HANDLE) can be grouped two categories: The first row of premises *(1a)* and *(1b)* are well-formedness conditions to assert non-escaping. The other two rows of premises type check the handler body *(2)* and the handled program *(3)*. I will now work through the different premises starting from the last one, highlighting important aspects. Premise *(3)* types the handled program and brings a capability of type $\mathsf{Eff}[S, T]\,\hat{}\,\{\mathbf{cap}\}$ into scope. Premise *(2)* types the body of the handler. It not only binds the argument of the effect operation y, but also the continuation, to which we assign the type $(\forall (T)\, R)\,\hat{}\,\{\mathbf{cap}\}$. The rule conservatively use $\{\mathbf{cap}\}$ as the capture set of the continuation in order to ensure the continuation cannot be accessed outside of the handler body. In particular, it could do so by being passed as an argument to another handler, by being passed as an argument to itself, or by being (indirectly) returned from the handler body. Finally, to guarantee that all accesses to the handler capability are well-scoped premises *(1a)* and *(1b)* require that the root capability **cap** is not included in $\mathrm{cs}(S)$ (and $\mathrm{cs}(R)$ respectively). This has an interesting consequence: the capture sets of S and R need to be concrete capture sets. This restriction lets us rule out programs such as:

$$\begin{array}{c}\textbf{handle}\,\mathsf{x} = v \textbf{ in } \lambda\left(y\right)\, \mathsf{x}.\textbf{do}\ y \longrightarrow \\ \textbf{handle}\,\mathtt{@a13} = v \textbf{ in } \lambda\left(y\right)\, \mathtt{@a13}.\textbf{do}\ y \longrightarrow \\ \lambda\left(y\right)\, \mathtt{@a13}.\textbf{do}\ y\end{array}$$

where x is unbound after reduction. In addition to restricting the answer type R, we also restrict the argument type S. The motivation for this is more subtle. Let us assume the following example adapted from Biernacki et al. [2020]:

$$\begin{array}{l}\textbf{handle}\,\mathsf{x} : \mathsf{Eff}[(\forall(\mathsf{Unit})\,\mathsf{Unit})\,\hat{}\,\{\mathbf{cap}\}, \mathsf{Unit}] = \lambda\left(f\,k\right)\, f() \textbf{ in} \\ \quad \textbf{handle}\,\mathsf{y} = h \textbf{ in} \\ \quad\quad \mathsf{x}.\textbf{do}\ \lambda()\ \mathsf{y}.\textbf{do}\,()\end{array}$$

The example reduces as follows.

$$
\begin{array}{l}
\textbf{handle}\,\texttt{@a13} : \mathsf{Eff}[\mathsf{Unit}\hat{}\{\mathbf{cap}\}\mathsf{Unit}, \mathsf{Unit}] = \lambda\left(f\ k\right)\ f()\ \textbf{in} \\
\quad \textbf{handle}\,\texttt{@4f1} = h\ \textbf{in} \\
\quad\quad \texttt{@a13.}\textbf{do}\,(\lambda\,()\ \texttt{@4f1.}\textbf{do}\,()) \\
\quad \longrightarrow \\
[f := \ldots, l := \ldots](f\,()) \\
\quad \longrightarrow \\
(\lambda\,()\ \texttt{@4f1.}\textbf{do}\,())\,() \\
\quad \longrightarrow \\
\texttt{@4f1.}\textbf{do}\,()
\end{array}
$$

This leads to an undelimited, i.e., unhandled, effect call to the label `@4f1`. To avoid this, we need to rule out the possibility that lambda abstractions closing over capabilities at the call site can be passed to effect operations. By requiring that the capture set on A needs to be concrete, we rule out the type of

$$\mathsf{Eff}[(\forall(\mathsf{Unit})\ \mathsf{Unit})\hat{}\{\mathbf{cap}\}, \mathsf{Unit}]$$

instead we would need to give the more precise type $\mathsf{Eff}[(\forall(\mathsf{Unit})\ \mathsf{Unit})\hat{}\{\mathsf{y}\}, \mathsf{Unit}]$. However, this is again ruled out, since it is not well-formed in the outer typing context. y is not bound at the handling site of x.

Conclusion

Capture sets allow reasoning about capability safety: we can ensure all effects are handled simply by establishing that capabilities do not leave their corresponding effect handlers, without equipping the language with an additional effect system. Capture sets also allow us reasoning about the effects used by a function. Inspecting the capture set on the type of a function value, we can conclude which effects can potentially be used by this function and in particular, which effects *cannot* be used.

Syntax

$l ::= \texttt{@a13}, \texttt{@4f1}, \ldots$	runtime labels
$v ::= \ldots \mid l$	label value
$c ::= \ldots \mid l$	label capture
$t ::= \ldots \mid \textbf{handle}\, l : \mathsf{Eff}[T, T] = \lambda\,(x\, x)\ t\ \textbf{in}\ t$	effect handler
$\Sigma ::= \emptyset \mid \Sigma,\ l : \mathsf{Eff}[T, T]$	signature environment

$$\mathrm{fv}(l) = \{l\} \qquad\qquad \mathrm{fv}(\textbf{handle}\, l : T\ \textbf{in}\ t) = \mathrm{fv}(t) \setminus l$$

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

SC-LABEL
$$\Gamma \vdash \{l\} <: \{\textbf{cap}\}$$

Type assignment $\boxed{\Gamma \mid \Sigma\ \vdash t : T}$

HANDLE-L
$$\frac{\begin{array}{c} \textit{(1a)}\quad \textbf{cap} \notin \mathrm{cs}(S) \qquad \textit{(1b)}\quad \textbf{cap} \notin \mathrm{cs}(T) \\ \textit{(2)}\quad \Gamma,\ y : S,\ k : (\forall (x : S)\, R)\,\hat{}\,\{\textbf{cap}\} \mid \Sigma\ \vdash s : R \\ \textit{(3)}\quad \Gamma \mid \Sigma,\ l : \mathsf{Eff}[S, T]\ \vdash t : R \end{array}}{\Gamma \mid \Sigma\ \vdash \textbf{handle}\, l : \mathsf{Eff}[S, T] = \lambda\big(y\, k\big)\ s\ \textbf{in}\ t : R}$$

LABEL
$$\frac{l : \mathsf{Eff}[S, T] \in \Sigma}{\Gamma \mid \Sigma\ \vdash l : \mathsf{Eff}[S, T]\,\hat{}\,\{l\}}$$

Reduction $\boxed{t \longrightarrow t}$

BETA-HANDLE-INTRO
$$\frac{l \text{ fresh}}{\textbf{handle}\, x = \lambda\big(y\, k\big)\ s\ \textbf{in}\ t \longrightarrow \textbf{handle}\, l = \lambda\big(y\, k\big)\ s\ \textbf{in}\ [x \mapsto l]\, t}$$

BETA-HANDLE
$$\textbf{handle}\, l = h\ \textbf{in}\ w \longrightarrow w$$

CONTEXT-HANDLE
$$\frac{\mathsf{E} = \textbf{handle}\, l = \lambda\big(y\, k\big)\ s\ \textbf{in}\ \mathsf{E}'}{\mathsf{E}[l.\textbf{do}\ v] \longrightarrow [k := \lambda(z)\, \mathsf{E}[z], y := v]\, s}$$

$$\mathsf{E} ::= \ldots \mid \textbf{handle}\, l = h\ \textbf{in}\ \mathsf{E} \mid \mathsf{E}.\textbf{do}\ t \mid l.\textbf{do}\ \mathsf{E}$$

Figure 3.12: Operational semantics of CF$_{<:}$ extended with effect handlers. The *highlighted* portions are merely a proof device to ensure soundness of the extension.

4 Boxing Capabilities: $\mathsf{CC}_{<:\square}$

4.1 Introduction

At the end of Chapter 2, we have seen the subtle issue with extending SCC to support universal polymorphism: the type system either needs to be aware that there is a capture set "under" a type variable, or type variables must only range over pure types, ones with an empty capture set.

In Chapter 3, we have seen $\mathsf{CF}_{<:}$, which extends SCC with type variables ranging over arbitrary types and allows type variables to appear in capture sets.

In this chapter, I present $\mathsf{CC}_{<:\square}$, a further result of the collaboration with Martin Odersky, Jonathan Brachthäuser, Ondřej Lhoták, and Edward Lee. $\mathsf{CC}_{<:\square}$ takes the second approach to extending SCC with universal polymorphism. Since type variables ranging over pure types on their own are not very expressive, we will see that $\mathsf{CC}_{<:\square}$ allows injecting impure capturing types into pure types. In comparison to $\mathsf{CF}_{<:}$, $\mathsf{CC}_{<:\square}$ supports more ergonomic types and forms the formal basis for the Scala implementation of Capture Tracking. Additionally, in this chapter I present examples which illustrate the practical usability of a Capture Tracking implementation based on $\mathsf{CC}_{<:\square}$; the examples were checked with the capture checker prototype 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. As we have said in the publication on which this chapter is based [Boruch-Gruszecki et al. 2023], 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 [Odersky et al. 2021; Brachthäuser et al. 2022]. Doing so avoids having 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 terms and types when they enter a generic context and later unboxing them on every use site [Brachthäuser et al. 2022].

Once again, whereas many of the examples motivating Capture Tracking describe applications in effect checking, the core formal system of $\mathsf{CC}_{<:\square}$ does not mention effects. The effect domains are intentionally kept open since they are orthogonal to the aims of the core system. Effects could be exceptions, file operations or region allocations, but also algebraic effects, IO, or any sort of monadic effects. More advanced control effects usually require adding continuations to the operational semantics, or using an implicit translation to the continuation monad. In short, capabilities can restrict what effects can be performed at any point in the program, but they by themselves don't perform an effect [Marino and Millstein 2009b; Gordon 2020; Liu 2016; Brachthäuser et al. 2020a; Osvald et al. 2016]. To actually perform an effect in the program, a library or a runtime system is needed. Both would need to be added as an extension of $\mathsf{CC}_{<:\square}$. Since $\mathsf{CC}_{<:\square}$ is intended to work with all such effect extensions, we refrain from adding a specific application to its core operational semantics.

Later, I introduce an extension of $\mathsf{CC}_{<:\square}$ to demonstrate how to properly enforce scoping of capabilities. The extension adds just enough primitives to $\mathsf{CC}_{<:\square}$ so that ill-scoped terms could get stuck, and then proceeds to show that all such terms are ruled out by the type system.

The version of $\mathsf{CC}_{<:\square}$ presented here evolved from a system which 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 as arguments. 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 $\mathsf{CC}_{<:\square}$. At the time, the metatheory of the proposed system was not worked out yet and the progress and preservation properties were left as conjectures. Additionally, there are some minor differences in the operational semantics, which were necessary to make the progress property hold.

The rest of this chapter is organized as follows. Section 4.2 explains and motivates key aspects of capture tracking present in $\mathsf{CC}_{<:\square}$. Section 4.3 presents $\mathsf{CC}_{<:\square}$ itself. Section 4.4 lays out its meta-theory. Section 4.5 illustrates the expressiveness of typing disciplines based on the calculus in examples. Section 4.6 discusses the need for $\mathsf{CC}_{<:\square}$ to include boxes as a core primitve. Section 4.7 extends $\mathsf{CC}_{<:\square}$ with a primitive scoped capability. Finally, Section 4.8 concludes.

4.2 Key Aspects of Capture Tracking in $\mathsf{CC}_{<:\square}$

$\mathsf{CC}_{<:\square}$ builds on top of SCC, featuring all the aspects of Capture Tracking discussed in Section 2.1: the hierarchy of capabilities, dependent function types, capture-checking closures using the variables of their bodies, relating capture sets with subcapturing, and using capture checking to ensure well-scoped access to local capabilities. To those, $\mathsf{CC}_{<:\square}$ adds one new aspect: *capture tunneling.*

All examples are written in an experimental language extension of Scala 3 [Scala 2022b] and were compiled with the prototype implementation of a capture checker [Scala 2022c].

4.2.1 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[1]:

```
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: Int ->{ct} String
def y: Logger^{fs}
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 be the type of `p`? Maybe surprisingly, it will be typed as:

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

That is, the outer capture set is empty and it neither mentions `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, $\mathsf{CC}_{<:\square}$ 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.

[1]This class is covariant in both A and B, as denoted by the pluses.

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, as shown in the following example.

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

The mapFirst function takes a Pair and a mapping function and applies the latter to the first element of the former. Class Pair merely retains both its constructor arguments in immutable fields. Thanks to 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]^{**cap**}, 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]^{**cap**} as the result type, which is an unacceptably inaccurate type.[2]

Section 4.3 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. Boxed values need an explicit unbox operation before they can be accessed, and that unbox operation charges the capture set of the environment. If the unbox operation is part of a closure, the unboxed type's capture set will contribute to the captured variables of that closure. The need for such a mechanism is explained in more detail in Section 4.6.

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. In this particular example, mapFirst operates only on generic types and no box or unbox operations need to be inserted; *how* such operations are inserted is discussed in Section 4.5.

[2]That is: assuming that we keep capture sets as sets of term variables. If we allow capture sets to contain *type* variables as in $\mathsf{CF}_{<:}$, the result can be typed as Pair[C,B]^{C}.

Escape Checking

There is now an additional consideration when checking that local capabilities are not accessed outside of their scope, compared to the concerns already mentioned in Section 2.1.5. One also needs to prevent returning or assigning a closure with a local capability in an argument of a parametric type. The following example illustrates the idea.

```
val sneaky = usingFile("out", os => Pair(() => os.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 `{cap}`: since `f` is no longer in scope we need to widen the type to a supertype that does not mention it (*cf.* the explanation of avoidance in Section 4.3.3). This causes an error, as the universal capability is not permitted to be in the unboxed form of the return type (*cf.* the precondition of (UNBOX) in Figure 4.2).

4.3 The $\mathsf{CC}_{<:\Box}$ Calculus

The syntax of $\mathsf{CC}_{<:\Box}$ is given in Figure 4.1. In short, it describes a dependently typed variant of System $\mathsf{F}_{<:}$ in monadic normal form (MNF) with capturing types and boxes.

Boxes. $\mathsf{CC}_{<:\Box}$ type variables X range over shape types only, not 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 $C \multimapinv x$, where C is a capture set of the underlying type of x. We have seen in Section 4.2 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 $\mathsf{F}_{<:}$. $\mathsf{CC}_{<:\Box}$ extends a (perhaps *the*) standard system with the two principal forms of polymorphism, subtyping and universal, just as $\mathsf{CF}_{<:}$ did.

The specific challenges posed by the combination of universal polymorphism and Capture Tracking are addressed in $\mathsf{CC}_{<:\Box}$ by restricting type variables to shape types and mediating between shape types and regular types via box and unbox operations.

Capture Sets. $\mathsf{CC}_{<:\Box}$ capture sets C are finite term variable sets $\{\overline{x}\}$, just as in SCC and as opposed to $\mathsf{CF}_{<:}$ capture sets, which can also contain type variables.

Capture sets of closures are determined using the cv function, in contrast to SCC using the free variables fv instead. First, I define cv; then, I contrast it with fv.

Variable	$x, y, z, \mathbf{cap}$		
Type Variable	X, Y, Z		
Value	v, w	$::=$	$\lambda(x : T)\, t \mid \lambda[X <: S]\, t \mid \Box\, x$
Answer	a	$::=$	$v \mid x$
Term	s, t	$::=$	$a \mid x\, y \mid x\, [S] \mid \mathbf{let}\, x = s\, \mathbf{in}\, t \mid C \multimap x$
Shape Type	S	$::=$	$X \mid \top \mid \forall(x : U) T \mid \forall[X <: S] T \mid \Box\, T$
Type	T, U	$::=$	$S \mid S \,\hat{}\, C$
Capture Set	C	$::=$	$\{\overline{x}\}$
Typing Context	Γ, Δ	$::=$	$\emptyset \mid \Gamma, X <: S \mid \Gamma, x : T \qquad \mathbf{if}\; x \neq \mathbf{cap}$

Figure 4.1: Syntax of $\mathsf{CC}_{<:\Box}$

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

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

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 $\mathbf{let}\, x = v\, \mathbf{in}\, t$, 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, while unboxing a term presents variables that were obscured when boxing. The third rule is motivated by the case where a variable x is bound to a value v; then we do not want to count the captured variables of v if x is either boxed or not mentioned at all in the let body. The intuition behind this rule is that such variables would naturally be disregarded if $\mathsf{CC}_{<:\Box}$ was not in MNF.[3]

Figure 4.2 presents the typing rules and operational semantics of $\mathsf{CC}_{<:\Box}$. In the following sections, I separately discuss the rules governing subcapturing, subtyping, typing and reduction.

4.3.1 Subcapturing

Subcapturing in $\mathsf{CC}_{<:\Box}$ is defined by the very same rules as in SCC see Section 2.2.2 for a discussion of the rules. Just like in SCC, we can establish that $\{\mathbf{cap}\}$ and $\{\}$ are respectively the top and bottom capture sets.

Proposition 4.1. *If C is well-formed in Γ, then $\Gamma \vdash \{\} <: C <: \{\mathbf{cap}\}$.*

A proof is enclosed in an appendix.

Proposition 4.2. *The subcapturing relation $\Gamma \vdash \_ <: \_$ is a preorder.*

Proof. We can show that transitivity and reflexivity are admissible. $\Box$

4.3.2 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 $S \equiv S \wedge \{\}$, 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.

4.3.3 Typing

The typing rules for type abstractions are close to System $\mathsf{F}_{<:}$, with differences only to account for capturing types. Typing rules are again close to System $\mathsf{F}_{<:}$, with differences to account for capture sets.

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

[3]Note that it is *boxing* which makes using cv necessary (as opposed to using fv). MNF alone does not require using cv.

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

$$\textsc{sc-elem}\ \frac{x \in C}{\Gamma \vdash \{x\} <: C} \qquad \textsc{sc-set}\ \frac{\overline{\Gamma \vdash \{x_i\} <: C}^{\,i}}{\Gamma \vdash \{\overline{x_i}^{\,i}\} <: C} \qquad \textsc{sc-var}\ \frac{x : S\,\hat{}\,C' \in \Gamma \qquad \Gamma \vdash C' <: C}{\Gamma \vdash \{x\} <: C}$$

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

$$\textsc{refl}\ \Gamma \vdash T <: T \qquad \textsc{top}\ \Gamma \vdash S <: \top \qquad \textsc{tvar}\ \frac{X <: S \in \Gamma}{\Gamma \vdash X <: S} \qquad \textsc{trans}\ \frac{\Gamma \vdash T_1 <: T_2 \quad \Gamma \vdash T_2 <: T_3 \quad \Gamma \vdash T_2\ \mathbf{wf}}{\Gamma \vdash T_1 <: T_3}$$

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

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

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

$$\textsc{var}\ \frac{x : S\,\hat{}\,C \in \Gamma}{\Gamma \vdash x : S\,\hat{}\,\{x\}} \qquad \textsc{abs}\ \frac{\Gamma, x : U \vdash t : T \qquad \Gamma \vdash U\ \mathbf{wf}}{\Gamma \vdash \lambda(x : U)\, t : (\forall(x : U)\, T)\,\hat{}\,\mathrm{cv}(t)\backslash x} \qquad \textsc{app}\ \frac{\Gamma \vdash x : (\forall(z : U)\, T)\,\hat{}\,C \qquad \Gamma \vdash y : U}{\Gamma \vdash x\, y : [z := y]T}$$

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

$$\textsc{tabs}\ \frac{\Gamma, X <: S \vdash t : T \qquad \Gamma \vdash S\ \mathbf{wf}}{\Gamma \vdash \Lambda[X <: S]\, t : (\forall[X <: S]\, T)\,\hat{}\,\mathrm{cv}(t)} \qquad \textsc{tapp}\ \frac{\Gamma \vdash x : (\forall[X <: S]\, T)\,\hat{}\,C}{\Gamma \vdash x[S] : [X := S]T}$$

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

Reduction $\boxed{t \longrightarrow t'}$

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

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

Figure 4.2: $\mathsf{CC}_{<:\Box}$ typing rules and operational semantics.

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

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 fact that the closure is consumed by the application. Rule (TAPP) is analogous.

Aside: A more conventional version of (TAPP) would be the following one.

$$\textsc{tapp'}\quad \frac{\Gamma \vdash x : (\forall[X <: S]'T)\,\hat{}\,C \qquad \Gamma \vdash S <: S'}{\Gamma \vdash x\,[S] : [X := S]\,T}$$

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 we can create a boxed type with {**cap**} as its capture set through subsumption, we cannot unbox values of this type. This property is fundamental for ensuring scoping of capabilities: recall that in Section 2.1.5 I have discussed how we can do so via a scheme which relies on checking if particular types only capture capabilities bound in the current context. Boxing a capability temporarily makes it not count as captured and forces the exact same check to be carried out when the capability is unboxed, which simultaneously enables capture tunneling *and* can serve as the basis for ensuring that capability access is well-scoped.

Avoidance.

In $\mathsf{CC}_{<:\square}$, just like in SCC, there is always a most specific avoiding type for a (LET).

Proposition 4.3. *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* $x \notin \mathrm{fv}(T_3)$.

A proof is attached in an appendix.

4.3.4 Well-Formedness

$\mathsf{CC}_{<:\square}$ well-formedness $\Gamma \vdash T\ \textbf{wf}$ 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 root capability **cap** in their capture sets. I present the well-formedness rules in Figure 4.3.

Type well-formedness $\boxed{\Gamma \vdash C\ \mathbf{wf}}$ $\boxed{\Gamma \vdash T\ \mathbf{wf}}$

$$\textsc{wf-cset}\ \frac{C \subseteq \mathrm{dom}(\Gamma) \cup \{\mathbf{cap}\}}{\Gamma \vdash C\ \mathbf{wf}} \qquad \textsc{wf-tvar}\ \frac{X <: S \in \Gamma}{\Gamma \vdash X\ \mathbf{wf}} \qquad \textsc{wf-fun}\ \frac{\Gamma \vdash U\ \mathbf{wf} \qquad \Gamma, x : U \vdash T\ \mathbf{wf}}{\Gamma \vdash \forall(x : U) T\ \mathbf{wf}}$$

$$\textsc{wf-capt}\ \frac{\Gamma \vdash C\ \mathbf{wf} \qquad \Gamma \vdash S\ \mathbf{wf}}{\Gamma \vdash S\,\hat{}\,C\ \mathbf{wf}} \qquad \textsc{wf-boxed}\ \frac{\Gamma \vdash T\ \mathbf{wf}}{\Gamma \vdash \Box T} \qquad \textsc{wf-tfun}\ \frac{\Gamma \vdash S\ \mathbf{wf} \qquad \Gamma, X <: S \vdash T\ \mathbf{wf}}{\Gamma \vdash \forall[X <: S] T\ \mathbf{wf}} \qquad \textsc{wf-top}\ \Gamma \vdash \top\ \mathbf{wf}$$

Figure 4.3: Well-formedness rules of $\mathsf{CC}_{<:\Box}$.

4.3.5 Operational Semantics

Similarly to SCC, $\mathsf{CC}_{<:\Box}$ operational semantics are defined by a small-step reduction relation which reduces terms under let-bindings and looks up the values to which variables are bound in the context of the redex.

The first three rules — (APPLY), (TAPPLY), (OPEN) — rewrite simple redexes: applications, type applications and unboxings. Rule (APPLY) is identical to the one from SCC, while the other two rules reduce new term forms. All three rules look up a variable in the enclosing store and proceed based on the found value.

The last two rules, (RENAME) and (LIFT) are administrative in nature and identical to the ones in SCC. Recall that 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 4.4. *Evaluation is deterministic. If $t \longrightarrow u_1$ and $t \longrightarrow u_2$, then $u_1 = u_2$.*

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

4.4 Metatheory

I now present the metatheoretic properties of $\mathsf{CC}_{<:\Box}$. For the most part, they are exactly analogous to the properties of SCC. The latter is chronologically younger: it is a variant of $\mathsf{CC}_{<:\Box}$ without universal type polymorphism and boxes. The proofs for all the lemmas and theorems stated in this section are provided in an appendix. The Progress and Preservation Theorems and the Capture Prediction Lemma for the calculus were also mechanized by Fourment and Xu [2023].

As usual for the systems in this dissertation, the metatheory of $\mathsf{CC}_{<:\Box}$ follows the Barendregt convention: we only consider typing contexts where all variables are unique, i.e., for all contexts of the form $\Gamma, x : T$ we have $x \notin \mathrm{dom}(\Gamma)$.

Figure 4.4 shows the definition of matching contexts (identical to the one in SCC).

$$\frac{\Gamma \vdash v : T \qquad \Gamma, x : T \vdash \sigma \sim \Delta}{\Gamma \vdash \textbf{let}\, x = v\, \textbf{in}\, \sigma \sim x : T, \Delta} \qquad\qquad \Gamma \vdash [\,] \sim \cdot$$

Figure 4.4: Matching environment $\boxed{\Gamma \vdash \sigma \sim \Delta}$

Recall that 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. The four lemmas relating the store and evaluation contexts to typing hold in $\mathsf{CC}_{<:\Box}$ just as they did in SCC.

Definition 4.1 (Evaluation Context Typing)**.** *Evaluation context η can be typed as $U \Rightarrow T$ in Γ, written $\Gamma \vdash \eta : U \Rightarrow T$, iff for all t such that $\Gamma \vdash t : U$ we have $\Gamma \vdash \eta[\, t\,] : T$.*

Lemma 4.1 (Evaluation Context Typing Inversion)**.** *$\Gamma \vdash \eta[\, s\,] : T$ implies that for some U we have $\Gamma \vdash \eta : U \Rightarrow T$ and $\Gamma \vdash s : U$.*

Lemma 4.2 (Evaluation Context Reification)**.** *If both $\Gamma \vdash \eta : U \Rightarrow T$ and $\Gamma \vdash s : U$, then $\Gamma \vdash \eta[\, s\,] : T$.*

Lemma 4.3 (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 4.4 (Store Context Reification)**.** *If both $\Gamma, \Delta \vdash t : T$ and $\Gamma \vdash \sigma \sim \Delta$, then $\Gamma \vdash \sigma[\, t\,] : T$.*

The Preservation and Progress Theorems are stated exactly the same as for SCC. Compared to the System $\mathsf{F}_{<:}$ theorems, the ones presented here have adjustments only to account for MNF.

Theorem 4.1 (Preservation)**.** *If we have $\Gamma \vdash \sigma \sim \Delta$ and $\Gamma, \Delta \vdash t : T$, then $\sigma[\, t\,] \longrightarrow \sigma[\, t'\,]$ implies that $\Gamma, \Delta \vdash t' : T$.*

Definition 4.2 (Proper Configuration)**.** *A term form $\sigma[\, t\,]$ is a* proper configuration *if t is not of the form $\textbf{let}\, x = v\, \textbf{in}\, t'$.*

Theorem 4.2 (Progress)**.** *If $\vdash \sigma[\, t\,] : T$ and $\sigma[\, t\,]$ is a proper configuration, then either t is an answer a, or $\sigma[\, t\,] \longrightarrow \sigma[\, t'\,]$ for some t'.*

Capture sets and captured variables

I now revisit the lemmas relating the variables of terms to the capture sets assigned to terms by typing.

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 4.5 (Capture Prediction for Answers)**.** *If* $\Gamma \vdash \sigma[\,a\,] : S \wedge C$, *then* $\Gamma \vdash \mathrm{cv}(\sigma[\,a\,]) <: C$.

In contrast to SCC, this lemma uses cv and not fv, to account for $\mathsf{CC}_{<:\Box}$ featuring boxes.

Just as for SCC, this lemma can be extended to apply to unreduced terms $\sigma[\,t\,]$. Recall that such terms can mention and use capabilities which are not reflected in the final capture set at all, since capture sets assigned to terms describe the capabilities captured by the value the term reduces to. Once again, this is formally expressed by the following property..

Lemma 4.6 (Capture Prediction for Terms)**.** *Let* $\vdash \sigma \sim \Delta$ *and* $\Delta \vdash t : S \wedge C$. *Then* $\sigma[\,t\,] \longrightarrow^* \sigma[\,\sigma'[\,a\,]\,]$ *implies that* $\Delta \vdash \mathrm{cv}(\sigma'[\,a\,]) <: C$.

4.4.1 Predicting Used Capabilities

In this section, I develop an additional correctness criterion for $\mathsf{CC}_{<:\Box}$: a theorem that uses capture sets to predict what capabilities may be used during the reduction of a term. Since the ability to perform effects is mediated by capabilities in capability-safe systems, predicting what capabilities may be used by terms gives a formal basis for reasoning about the *authority* of programs to perform side-effectful operations [Miller 2006; Drossopoulou et al. 2016]. As I will discuss later, this theorem is also an important correctness criterion for boxing. It is much less interesting in a system without boxing, explaining its absence from metatheories of systems presented in previous chaptes.

If we want to reason about what capabilities are used, we need to have a concept of primitive capabilities which must be tracked, not unlike how STLC needs base types to make its correctness theorem non-vacuous [Pierce 2002]. While object capabilities come in many forms, for our current purposes it suffices to consider capabilities that exist for the entire duration of the program, such as a capability to access the filesystem or the standard output. Within core $\mathsf{CC}_{<:\Box}$, we can simply designate an outer fragment of the store as the *platform* context Ψ, which introduces well-behaved primitive capabilities:[4]

$$\Psi \quad ::= \quad [\,] \quad | \quad \textbf{let}\, x = v\, \textbf{in}\, \Psi \qquad \textbf{if}\ \mathrm{fv}(v) = \{\}$$

The operational semantics of the capabilities in Ψ are defined by the values v. The values need to be closed, since otherwise the capabilities would depend on other capabilities and would not be primitive. Since Ψ binds capabilities, their capture set should be $\{\mathbf{cap}\}$:

Definition (Well-typed program)**.** *A term* $\Psi[\,t\,]$ *is a well-typed program if we have* $\Delta \vdash t : T$ *for some* Δ *such that* $\vdash \Psi \sim \Delta$ *and for all* $x \in \Delta$ *there exists a shape type* S *such that* $x : S \wedge \{\mathbf{cap}\} \in \Delta$.

It is now possible to state an intermediate lemma necessary to proving the desired correctness criterion:

[4]This is another advantage of the MNF "restriction". Since variables bound in the surrounding store are never substituted and a function applied to such a variable will reduce, we can faithfully model capabilities as such variables without having to introduce labels as a separate concept, as we had to do in $\mathsf{CF}_{<:}$ (Section 3.3).

Lemma 4.7 (Program authority preservation)**.** *Let* $\Psi[\,t\,] \longrightarrow \Psi[\,t'\,]$, *where* $\Psi[\,t\,]$ *is a well-typed program. Then* $\mathrm{cv}(t')$ *is a subset of* $\mathrm{cv}(t)$.

I will now formally state what capabilities are used during evaluation. Since Ψ only binds abstractions, it makes sense to say a capability x is used if during evaluation we reduced an application form.

Definition (Used capabilities)**.**

$$\begin{array}{lcll}
\mathrm{used}(t_1 \longrightarrow t_2 \longrightarrow \cdots \longrightarrow t_n) & \triangleq & \mathrm{used}(t_1 \longrightarrow t_2) \cup \mathrm{used}(t_2 \longrightarrow \cdots \longrightarrow t_n) & \\
\mathrm{used}(\sigma[\,\eta[\,x\,y\,]\,] \longrightarrow \sigma[\,t\,]) & \triangleq & \{x\} & \\
\mathrm{used}(\sigma[\,\eta[\,x\,[S]\,]\,] \longrightarrow \sigma[\,t\,]) & \triangleq & \{x\} & \\
\mathrm{used}(t_1 \longrightarrow t_2) & \triangleq & \{\} & \textit{(\textbf{otherwise})}
\end{array}$$

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

All the definitions needed to state the correctness theorem are now in place.

Theorem 4.3 (Used capability prediction)**.** *Let* $\Psi[\,t\,] \longrightarrow^* \Psi[\,t'\,]$, *where* $\Psi[\,t\,]$ *is a well-typed program. Then the primitive capabilities used during the reduction are a subset of the authority of* t*:*

$$\{\, x \mid x \in \mathrm{used}(\Psi[\,t\,] \longrightarrow^* \Psi[\,t'\,]), x \in \mathrm{dom}(\Psi)\,\} \subseteq \mathrm{cv}(t)$$

4.4.2 Correctness of Boxing

Both Lemma 4.7 and Theorem 4.3 would be trivially true if $\mathrm{cv}(t)$ was just the free variables of t, since reduction typically does not add new free variables to a term. However, boxes allow preventing some captured free variables from appearing in capture sets. For instance, if we first box x and then pass it as an argument to f, the overall cv will not mention x:

$$\mathrm{cv}(\mathbf{let}\,y = \Box\,x\,\mathbf{in}\,f\;y) = \{f\}$$

Given this behavior, what is the correctness criterion for how we type box and unbox forms? Intuitively, it should not be possible to “smuggle in” a capability via boxes: a term’s capabilities should all be accounted for. By the Progress Theorem and a straightforward induction, we can prove that the cv of a term which boxes and immediately unboxes a capability accounts for the unboxed capability:

Proposition 4.5. *Let* $\vdash \sigma \sim \Delta$ *and* $t = (\textbf{\textit{let}}\,y = \Box\,x\,\textbf{\textit{in}}\,C \multimapinv y)$ *such that we have* $\Delta \vdash \eta[\,t\,] : T$ *for some* η *and* T. *Then* $\mathrm{cv}(t) = C$ *and we also have:*

$$\Delta \vdash \{x\} <: C$$

Speaking more generally, the fundamental function of boxes is that they allow *temporarily* preventing a captured free variable from affecting the cv of a term. The capability inside the box can still be used via the unbox form $C \multimap x$, but only at the cost of adding C, the "key" used to open the box, to the cv of the term. The correctness criterion for box and unbox forms is that the keys used to open boxes should account for the capabilities inside the box: a term should only be able to use capabilities that are accounted for by its cv, just as Lemma 4.7 and Theorem 4.3 show.

There is another aspect of boxing explained by these theorems: boxes can later be opened with unbox forms, shifting where capture sets appear. As an example, consider the following two lambdas, both of which may use fs (where $\mathsf{Proc} \triangleq \forall(x:\mathsf{Unit})\mathsf{Unit}$):

$$\begin{array}{ll} \mathsf{fs}:\mathsf{Fs}\,\hat{}\,\{\mathbf{cap}\} \vdash \lambda\big(f:\mathsf{Proc}\,\hat{}\,\{\mathsf{fs}\}\big)\; f\,() & :\forall(f:\mathsf{Proc}\,\hat{}\,\{\mathsf{fs}\})\mathsf{Unit} \\ \mathsf{fs}:\mathsf{Fs}\,\hat{}\,\{\mathbf{cap}\} \vdash \lambda\big(f:\Box\,\mathsf{Proc}\,\hat{}\,\{\mathsf{fs}\}\big)\;\mathbf{let}\,g = \{\mathsf{fs}\} \multimap f\;\mathbf{in}\;g\,() & :(\forall(f:\Box\,\mathsf{Proc}\,\hat{}\,\{\mathsf{fs}\})\mathsf{Unit})\,\hat{}\,\{\mathsf{fs}\} \end{array}$$

Figure 4.5: An example of boxes shifting what capture sets are charged with capabilities.

The first lambda's argument is a capability: a closure capturing fs. The lambda can invoke this closure without affecting its capture set. Meanwhile, the argument of the second lambda is *pure*: a box containing a closure capturing fs. The second lambda can still invoke its argument, but only after unboxing it, which charges its capture set with the fs capability.

Understanding that capture sets describe the authority of terms explains why it is sound for boxes to shift a capability from one capture set to another. To illustrate, let Γ bind the first closure from Figure 4.5 as $f_1:\forall(g:\mathsf{Proc}\,\hat{}\,\{\mathsf{fs}\})\mathsf{Unit}$ and the second closure as $f_2:(\forall(g:\Box\,\mathsf{Proc}\,\hat{}\,\{\mathsf{fs}\})\mathsf{Unit})\,\hat{}\,\{\mathsf{fs}\}$ and also bind an fs-capturing procedure as $p:\mathsf{Proc}\,\hat{}\,\{\mathsf{fs}\}$. Calling either f_1 or f_2 can use fs, which is reflected by cv even if the capture sets of f_1 and f_2 are different. In the first case, we have $\Gamma \vdash \mathrm{cv}(f_1\,p) <: \{p\} <: \{\mathsf{fs}\}$: we can elide f_1 from the capture set, but afterwards the smallest set we can widen to is {fs}. In the second case, we have $\Gamma \vdash \mathrm{cv}(\mathbf{let}\,p' = \Box\,p\;\mathbf{in}\;f_2\,p') = \{f_2\} <: \{\mathsf{fs}\}$: p is absent from the cv, but the smallest capture set to which we can widen $\{f_2\}$ is still {fs}. We correctly predict the authority of both terms.

When I refer to untracked closures, such as $f:(\forall(x:\mathsf{Unit})\mathsf{Unit})\,\hat{}\,\{\}$, as pure, I am also indirectly using the notion that a term's cv reflects its authority. What I mean is that such closures cannot be used to cause any effects on their own. Formally, when we reduce $f\,()$ to $[x := ()]\,t$, based on (ABS) we must have $\mathrm{cv}([x := ()]\,t) = \{\}$, i.e., the result is a term that cannot use any capabilities.

4.5 Examples

During the collaboration we have implemented a type checker for $\mathsf{CC}_{<:\Box}$ as an extension of the Scala 3 compiler, to enable experimentation with larger code examples. Notably, the 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. The presented implementation was developed almost entirely by Martin Odersky, but it and the examples typechecked with it are a significant part of the argument for the practicality of $\mathsf{CC}_{<:\square}$ as a formalism and Capture Tracking as an approach. Hence, I now present examples which demonstrate the usability of the implementation.

4.5.1 Church-Encoded Lists

In this section, I remain close to the core calculus by encoding lists using only functions; here, I still show the boxed types and boxing and unboxing operations that the compiler infers in gray, even though they are not in the source code.

Using the Scala prototype implementation of $\mathsf{CC}_{<:\square}$, the Böhm-Berarducci encoding [Böhm and Berarducci 1985] of a linked list data structure can be implemented and typed as follows. Recall from the analogous $\mathsf{CF}_{<:}$ example that in this encoding 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 eliding the capture set of the elements from the capture set of the containing list; something which was not the case in $\mathsf{CF}_{<:}$ (Section 3.3.1). 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:

The mapped function `f` may capture any capabilities, as documented by the capture set `{cap}` 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.

4.5.2 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[5] to allocate those objects on a stack as follows.[6]

```
class Pooled

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

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

[5] Remark that it sometimes makes sense to run a JVM *without* garbage collection: https://openjdk.org/jeps/318.
[6] For simplicity, this example is neither thread nor exception safe.

```
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^ => 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 **`cap`** would account for `{pooled}`, but we disallowed instantiating a type variable with `{`**`cap`**`}`. Having `withFreshPooled` defined as in the preceding example, the `pooledClosure` example is correctly rejected, while the following safe example is allowed.

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

4.5.3 Collections

In the following examples I show that a typing discipline based on $\mathsf{CC}_{<:\Box}$ 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 the current capture checking prototype [Scala 2022b].

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. First, I present 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. The type `IterableOnce[B]^` makes a subtle distinction: this collection may perform computation to produce elements of type `B`, and it may have captured capabilities to perform this computation as denoted by the "`^`". All these capabilities will have been used (and therefore discarded) by the time the resulting `List[B]` is produced. Of course, we could have left out the trailing "`^`"s, 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]^{it})
  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): Iterator[A]^{it}
  def map[B](f: A => B): Iterator[B]^{it, f}
  def flatMap[B](f: A => IterableOnce[B]^): Iterator[B]^{it, f}
  def ++[B >: A](xs: IterableOnce[B]^): Iterator[B]^{it, xs}
```

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, which matches the developer's intuitive understanding of reference patterns and signal information that is relevant in this context.

4.6 Why Boxes?

Boxed types and box/unbox operations are a key part of the calculus to make type abstraction work. This might seem suprising. After all, as long as the capture set is not the root capture set `{cap}`, one can always go from a capturing type to its boxed type and back by boxing and unboxing operations. So in what sense is this more than administrative ceremony? The key observation here is that an unbox operation $C \multimap x$ *charges the capture set of the environment* with the capture set C. If the unbox operation is part of a closure with body t then C will contribute to the captured variables $\mathrm{cv}(t)$ of the body and therefore to the capture set of the closure as a whole. In short, unbox operations propagate captures to enclosing closures (whereas, dually, box operations suppress propagation).

To see why this matters, assume for the moment a system with type polymorphism but without boxes, where type variables can range over capturing types but type variables are not themselves tracked in capture sets. Then the following is possible:

```
val framework
  : [X <: Any^] -> (x: X) -> (X -> Unit) -> Unit =
  = [X <: Any^] => (x: X) => (plugin: X -> Unit) => plugin(x)
```

The `framework` function combines the two sides of an interaction, with an argument `x` and an argument `plugin` acting on `x`. The interaction is generic over type variable `X`. Now instantiate `framework` like this:

```
val c: File^{cap}
val inst
  : (File^{c} -> Unit) -> Unit
  = framework[File^{c}](c)
```

This looks suspicious since `inst` now has a pure type with empty capture set, yet invoking it can access the `c` capability. Here is an example of such an access:

```
val writer
  : File^{c} -> Unit
  = (x: File^{c}) => x.write
inst(writer)
```

This invocation clearly executes an effect on the formal parameter `x`, which gets instantiated with `c`. Yet both `inst` and `writer` have pure types with no retained capabilities. Note that `writer` gets the necessary capability `{c}` from its argument, so the function itself does not retain capabilities in its environment, which makes it pure. It is difficult to see how a system along these lines could be sound. At the very least it would violate the Capture Prediction Lemma (Lemma 4.6).

Boxing the bound of `X` and adding the required box/unbox operations rejects the unsound instantiation. The definitions of `framework` and `inst` now become:

```
val framework
  : [X <: □ Any^] -> (x: X) -> (X -> Unit) -> Unit =
  = [X <: □ Any^] => (x: X) => (plugin: X -> Unit) => plugin(x)

val inst
  : (□ File^{c} -> Unit) -> Unit
  = framework[□ File^{c}](□ c)
```

Now any attempt to invoke `inst` as before would lead to an error:

```
val writer
  : (□ File^{c}) ->{c} Unit
  = (x: File^{c}) => ({c} ⟜ x).write
inst(writer)  // error
```

Indeed, `writer`, the argument to `inst`, now has the type

```
(□ File^{c}) ->{c} Unit
```

because the unbox operation in the lambda's body charges the closure with the capture set `{c}`. Therefore, the argument is now incompatible with `plugin`'s formal parameter type

```
(□ File^{c}) -> Unit
```

which is a pure function type.

This example shows why one cannot simply drop boxes and keep everything else unchanged. But one could also think of several other possibilities:

One alternative is to drop boxes, but keep the stratification of shape types and full types. Type variables would still be full types but not shape types. Such a system would certainly be simpler but it would also be too restrictive to be useful in practice. For instance, it would not be possible to invoke a polymorphic constructor that creates a list of functions that capture some capability.

In summary, a system with boxes turned out to lead to the best ergonomics of expression among the alternatives we considered. The core property of boxes is that unboxing charges the environment with the capture set of the unboxed type and thus allows to correctly recover captured references in a box without having to propagate these captures into the types of polymorphic type constructors. So in a sense, the conclusion is that one can always unbox (as long as the capture set is not the universal one), but it does not come for free.

Xu and Odersky [2023] show that boxed types and boxing and unboxing operations can be inferred. That paper presents an algorithmic type system that inserts boxed type constructors around capturing type arguments and inserts box and unbox operations as needed in the terms accessing values of these type arguments. As is typical, the algorithmic type system is significantly more involved than the declarative system presented in this chapter.

One can also turn that around. If we have a sound system with type variables (for instance by inserting implicit boxed types and box/unbox operation in the way our implementation works), then it is possible to define box and unbox as library operations in the language, along the following lines:

```
class Box[T](elem: T)
def box[T](x: T): Box[T] = new Box[T]
def unbox[T](x: Box[T]): T = x.elem
```

This construction demonstrates that in essence, boxes can be seen as a mechanism to obtain sound polymorphism for capturing types. Once we have a such a system, the functionality of source-boxes can also be obtained by defining a parametric class (or an equivalent Church-encoding) with a constructor/destructor pair. That's why the implementation does not need to expose boxed types and primitive box and unbox operations in the source code: a construction like the one above is enough to simulate this functionality.

4.7 Scoped Capabilities

In this section I present an extension to $\mathsf{CC}_{<:\square}$ which illustrates how boxes can be used to ensure scoping of capabilities, using the scheme discussed back in Section 2.1.5. Figure 4.6 shows the extensions to the static semantics. The extension is minimal: we add a boundary form $\textbf{boundary}[S]\, x \Rightarrow t$, mirroring a Scala 3 feature [Scala 2022a]. The boundary form delimits a scope that can be broken out of by using the *break capability* $x : \mathsf{Break}[S]$; the form is parameterised by a type argument S which can be inferred in the implementation. A boundary is a more expressive version of a labeled block that can be returned from: it also allows returning across closure and function boundaries since the break capability is a first-class value that can be closed over and passed as an argument. The type system should disallow invoking the capability once the boundary is left, since intuitively at that point there is no scope to be broken out of. As I explained in Section 4.4.2, a variable x of boxed type can only be opened via an unbox form $C \mathrel{\circ\!\!-} x$ such that C accounts for the capability in the box. The plan is simple: we 1) ensure that all capabilities leaving the **boundary** scope are boxed and 2) ensure that the scoped capability cannot be accounted for by any variable other than itself. In this extension, the only way for a scoped capability to leak is by being directly returned from its scope, so it suffices to require in rule (BOUNDARY) that the result of a **boundary** form is pure. To illustrate, consider the following attempt to leak a scoped capability by returning a closure (where $\mathsf{Proc} \triangleq \forall(y : \mathsf{Unit})\, \mathsf{Unit}$):

$$\vdash \quad \textbf{boundary}[\ldots]\, x \Rightarrow \textbf{let}\, f = \lambda(y : \mathsf{Unit})\; x\,()\, \textbf{in}\, \square f \quad : \quad \square\mathsf{Proc}\,\hat{}\,\{\textbf{cap}\}$$

Since a boundary's result must be pure, we have no choice but to box the closure. Since x is not in scope outside of the boundary, the capture set under the box must be $\{\textbf{cap}\}$. Since no typing context accounts for $\{\textbf{cap}\}$, the box cannot be opened anymore and we are safe.

In a fully featured programming language, there are other channels for scoped capabilities to leak, e.g. via mutable state. With boxing, to make such channels sound it suffices to only allow pure values to pass through them. For instance, if we want to store a capability in mutable state, we need to box it; afterwards we can only use it in a typing context that accounts for the capabilities under the box. In more complicated scenarios, a capability may return to its scope after leaving it; such cases could occur, for instance, when we allow sending values between

Syntax and definitions

Term	t, s	$::=$	$\mathbf{boundary}[S]\, x \Rightarrow t \mid \ldots$
Shape type	S, R	$::=$	$\mathsf{Break}[S] \mid \ldots$

$$\mathrm{cv}(\mathbf{boundary}[S]\, x \Rightarrow t) = \mathrm{cv}(t) \setminus x$$

Subtyping

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

Typing

$$\textsc{Boundary}\quad \frac{\Gamma, x : \mathsf{Break}[S]\,\hat{}\,\{\mathbf{cap}\} \vdash t : S \qquad x \notin \mathrm{fv}(S)}{\Gamma \vdash \mathbf{boundary}[S]\, x \Rightarrow t : S} \qquad \textsc{Invoke}\quad \frac{\Gamma \vdash x : \mathsf{Break}[S]\,\hat{}\,\{\mathbf{cap}\} \qquad \Gamma \vdash y : S}{\Gamma \vdash x\, y : T}$$

Figure 4.6: Scoped Capability Extensions to the static rules of System $\mathsf{CC}_{<:\Box}$

threads and when we allow effect-polymorphic effect handlers [Leijen 2014; Biernacki et al. 2020]. Boxing has been shown to be sound and behave as expected in the latter scenario: the boxed capability can be unboxed once it is back in its scope, but not earlier [Brachthäuser et al. 2022]. Thus, while the extension we show is minimal, it presents all the formal foundations we need for ensuring scoping of capabilities.

4.7.1 Dynamic Semantics of Scoped Capabilities

Figure 4.7 shows the extensions to the dynamic semantics of $\mathsf{CC}_{<:\Box}$. We add new evaluation-time term forms; we explain them by inspecting the relevant evaluation rules. Rule (ENTER) reduces a term of the form $\sigma[\,\eta[\,\mathbf{boundary}[S]\, x \Rightarrow t\,]\,]$ to $\sigma[\,\mathbf{let}\, x = l_S\, \mathbf{in}\, \eta[\,\mathbf{scope}_{l_S}\ t\,]\,]$: entering a boundary binds the break capability l_S in the store and adds a *scope* form to the evaluation context. The break capability is a *label l* annotated with the boundary's return type, where a label represents a boundary's unique runtime identifier. The scope form $\mathbf{scope}_{l_S}\ t$ is a marker on the stack (formally represented the evaluation context), denoting where a boundary ends; all scopes are annotated with their corresponding labels. When the break capability is invoked, the term has the form $\sigma[\,\eta_1[\,\mathbf{scope}_{l_S}\ \eta_2[\,x\, y\,]\,]\,]$ and the evaluation context is split by a scope form into the part outside and inside the scope. Rule (BREAKOUT) reduces such terms to

Syntax and definitions

Label	l	$::=$	$@123 \mid @456 \mid \ldots$
Value	v, w	$::=$	$l_S \mid \ldots$
Term	t, s	$::=$	$\textbf{scope}_{l_S}\ t \mid \ldots$
Captured Reference	c	$::=$	$x \mid l$
Capture Set	C	$::=$	$\{\overline{c}\}$

$$\mathrm{cv}(\textbf{scope}_{l_S}\ t) = \mathrm{cv}(t) \qquad\qquad \mathrm{cv}(l_S) = \{l\}$$

Subcapturing Base subcapturing rules use c instead of x.

SC-LABEL
$$\Gamma \vdash \{l\} <: \{\textbf{cap}\}$$

Typing

LABEL
$$\Gamma \vdash l_S : \mathsf{Break}[S]\,\hat{}\,\{l\}$$

SCOPE
$$\frac{\Gamma \vdash t : S}{\Gamma \vdash \textbf{scope}_{l_S}\ t : S}$$

Reduction

$$\begin{array}{llll}
\sigma[\,\eta[\,\textbf{boundary}[S]\,x \Rightarrow t\,]\,] \longrightarrow \sigma[\,\textbf{let}\,x = l_S\,\textbf{in}\,\eta[\,\textbf{scope}_{l_S}\ t\,]\,] & \textbf{if}\ l \text{ fresh} & (\textsc{enter}) \\
\sigma[\,\eta_1[\,\textbf{scope}_{l_S}\ \eta_2[\,x\,y\,]\,]\,] \longrightarrow \sigma[\,\eta_1[\,y\,]\,] & \textbf{if}\ \sigma(x) = l_S & (\textsc{breakout}) \\
\sigma[\,\eta[\,\textbf{scope}_{l_S}\ a\,]\,] \longrightarrow \sigma[\,\eta[\,a\,]\,] & & (\textsc{leave})
\end{array}$$

where Eval context $\eta \ ::= \ \textbf{scope}_{l_S}\,[\,] \mid \ldots$

Figure 4.7: Operational semantics of $\mathsf{CC}_{<:\Box}$ extended with scoped capabilities.

$\sigma[\,\eta_1[\,y\,]\,]$, dropping the scope form together with the inner evaluation context. Once only an answer remains under the scope, rule (LEAVE) reduces $\sigma[\,\eta[\,\textbf{scope}_{l_S}\ a\,]\,]$ to $\sigma[\,\eta[\,a\,]\,]$. Typing ensures that after a boundary is left, its capability is never invoked; otherwise we could get stuck terms since the scope form needed by (BREAK) would be absent from the evaluation context.

4.7.2 Metatheory

If we start evaluation from a term well-typed according to the static typing rules (one that does not mention any labels or scope forms), the evaluation rules maintain an invariant: all break capabilities are well-scoped, and all scope labels are unique; maintaining this invariant is necessary to avoid getting stuck terms. We state this invariant formally and incorporate it into the main correctness theorems.

For the purposes of our metatheory (including this invariant), we understand labels as primitive capabilities provided by the "runtime" to the program; in particular, the cv of a closed term may now mention labels, which we understand as the primitive capabilities a program can access.

Definition (Captured variables of contexts)**.** *We extend* cv *to contexts by* $\mathrm{cv}([\,]) = \{\}$.

Definition (Proper program)**.** *A term is a* proper program *if it has the form* $\sigma[\,\eta[\,t\,]\,]$ *s.t.:*

- *for all* l *such that* $l \in \mathrm{cv}(\sigma[\,\eta[\,t\,]\,])$*:*
  - *there exists a unique* x *such that* $\sigma(x) = l_S$ *for some* S
  - *there exist unique* η_1 *and* η_2 *such that* $\eta = \eta_1[\,\textbf{scope}_{l_S}\,\eta_2\,]$ *for the same* S
  - *for the same* η_1 *we have* $l \not\in \mathrm{cv}(\eta_1)$
- *scope forms in* $\sigma[\,\eta[\,t\,]\,]$ *only occur in* η

Theorem 4.4 (Preservation)**.** *Let* $\Gamma \vdash \sigma \sim \Delta$ *and* $\Gamma, \Delta \vdash t : T$*, where* $\sigma[\,t\,]$ *is a proper program. Then* $\sigma[\,t\,] \longrightarrow \sigma[\,t'\,]$ *implies that* $\Gamma, \Delta \vdash t' : T$ *and that* $\sigma[\,t'\,]$ *is a proper program.*

Theorem 4.5 (Progress)**.** *If* $\vdash \sigma[\,t\,] : T$ *and* $\sigma[\,t\,]$ *is a proper program and a proper configuration, then either* t *is an answer* a *or* $\sigma[\,t\,] \longrightarrow \sigma[\,t'\,]$ *for some* t'.

In the base system, we needed Theorem 4.3 to demonstrate that boxes are typed correctly, since unboxing a capability could never lead to a stuck term. In this extension, unboxing an out-of-scope capability *can* lead to a stuck term, so we can demonstrate soundness of the boxing rules in a more direct way, by showing the classical progress and preservation theorems. In fact, Lemma 4.7 and Theorem 4.4 both employ an identical argument in the case for rule (OPEN).

Predicting used capabilities

We can understand labels as the primitive capabilities a program may access. This makes the situation more complicated than before, since primitive capabilities can now be created and dropped. This is entirely expected when taking the object capability perspective. For example, in Wyvern [Melicher et al. 2017] creating capabilities is a commonplace occurrence, since an object with mutable state counts as a capability. In systems where file handles are capabilities, a capability is created or dropped every time we open or close a file handle.

This means that when reasoning about what capabilities are used, we need to consider what capabilities were created or dropped. To account for this, we reason about *traces*: sets of events that occurred during evaluation.

Definition (Evaluation trace)**.**

$$\begin{array}{lcll}
\text{trace}(t_1 \longrightarrow t_2 \longrightarrow \cdots \longrightarrow t_n) & \triangleq & \text{trace}(t_1 \longrightarrow t_2) \cup \text{trace}(t_2 \longrightarrow \cdots \longrightarrow t_n) & \\
\text{trace}(\sigma[\,\eta[\,x\,y\,]\,] \longrightarrow s) & \triangleq & \{\mathbf{use}(l)\} & \textbf{\textit{if}}\ \sigma(x) = l_S \\
\text{trace}(\sigma[\,\eta[\,\textbf{\textit{boundary}}[S]\,x \Rightarrow t\,]\,] \longrightarrow s) & \triangleq & \{\mathbf{create}(l)\} & \textbf{\textit{if}}\ s = \sigma'[\,\eta[\,\textbf{\textit{scope}}_{l_S}\ t\,]\,] \\
\text{trace}(\sigma[\,\eta[\,\textbf{\textit{scope}}_{l_S}\ a\,]\,] \longrightarrow s) & \triangleq & \{\mathbf{drop}(l)\} & \\
\text{trace}(t_1 \longrightarrow t_2) & \triangleq & \{\} & \textbf{\textit{otherwise}}
\end{array}$$

We need three auxilliary functions.

Definition (Used, created, and gained capabilities)**.**

$$\begin{aligned}
\text{used}(t \longrightarrow^* s) &= \{x \mid \mathbf{use}(x) \in \text{trace}(t \longrightarrow^* s)\} \\
\text{created}(t \longrightarrow^* s) &= \{x \mid \mathbf{create}(x) \in \text{trace}(t \longrightarrow^* s)\} \\
\text{gained}(t \longrightarrow^* s) &= \{x \mid \mathbf{create}(x) \in \text{trace}(t \longrightarrow^* s), \mathbf{drop}(x) \notin \text{trace}(t \longrightarrow^* s)\}
\end{aligned}$$

The Program Authority Preservation Lemma is now stated slightly differently. First, we only consider break capabilities to be primitive. Second, programs can gain authority over new capabilities, but *only* by creating them and *only* until the capabilities are dropped. Typing already ensures that all break capabilities are tracked and labels are always "bound", so it is now unnecessary to separately define platform contexts and well-typed programs.

Lemma 4.8 (Program authority preservation)**.** *Let $t \longrightarrow t'$, where $\vdash t : T$. Then:*

$$\text{cv}(t') \subseteq \text{cv}(t) \cup \text{gained}(t \longrightarrow t')$$

Finally, we reformulate the Used Capability Prediction Theorem.

Theorem 4.6 (Used Capability Prediction)**.** *Let $t \longrightarrow^* t'$, where $\vdash t : T$. Then the primitive capabilities used during the evaluation are within the authority of t:*

$$\text{used}(t \longrightarrow^* t') \subseteq \text{cv}(t) \cup \text{created}(t \longrightarrow^* t')$$

4.8 Conclusion

$\mathsf{CC}_{<:\Box}$ is a formal system for tracking 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.

5 Polymorphism and Capture Tracking

In the previous two chapters we have seen $\mathsf{CF}_{<:}$ and $\mathsf{CC}_{<:\Box}$ two systems which take different approaches to extending SCC with universal type polymorphism. $\mathsf{CF}_{<:}$ allows type variables which range over capturing types, while $\mathsf{CC}_{<:\Box}$ restricts type variables to ranging over pure types and introduces boxed types, which allow capture sets to "tunnel" through seemingly pure types. Ultimately, it was $\mathsf{CC}_{<:\Box}$ which was chosen as the formal basis for the experimental Capture Tracking implementation in Scala 3, since it has distinct advantages when it comes to commonplace types assigned to data structures.

Still, Capture Tracking is an ongoing research project larger than this thesis, which is focused on the formal foundations behind the approach. The key goal of Capture Tracking is to find a way of tracking capabilities in types which is sufficiently ergonomic to make a real impact on how programs are written in the industry. In pursuit of this goal, we have developed more than 7 different versions of the base formal system, and we have built (and rebuilt) diverse extensions based on subtly different versions of the formalism. The real test Capture Tracking has to pass is actually being adopted in the industry, by people who genuinely find the approach intuitive. The only way to check if it passes this test is to make an informed guess about what features make the approach intuitive, implement a system based on these features, and validate the guess empirically, by using the implementation and promoting its use in the industry. It may yet turn out that in order for the approach to be sufficiently practical, some of the restrictions imposed by the $\mathsf{CC}_{<:\Box}$ approach will need to be lifted.

With this in mind, the rest of this chapter is organized as follows. First I discuss three different classes of definitions where the $\mathsf{CF}_{<:}$ approach to universal type polymorphism has advantages compared to $\mathsf{CC}_{<:\Box}$; these classes may turn out to be significant enough to warrant extensions to the implementation. Afterwards, I conclude by proposing a way to integrate the two approaches in a single system.

5.1 Deferred Closures

The defer statement is a facility in the Go language which allows deferring code to be executed at the end of the current function; it is the idiomatic way to "clean up" the local state, for instance by closing file handles. A hypothetical extension to Scala which adds the `defer` statement may look as follows.

```
def foo() = {
  val file1 = new File(...)
  defer { file1.close() }
  val file2 = new File(...)
  defer { file2.close() }
  // Both `file1` and `file2` will be closed when `foo` exits.
  ...
}
```

Compared to "manually" closing the file handles at the end of the function, the `defer` statement makes it easier to do the right thing in the presence of exceptions; in particular, both deferred blocks will execute no matter if one of them aborts or not. Compared to the Java try-with-resources pattern, the `defer` statement avoids unnecessary, unwieldy levels of nesting.

The `defer` statement, like most facilities dealing with control flow, can be implemented using an effect handler. The idea is to introduce a lexical scope with a capability that allows deferring closures until the scope's end; since the closures are executed for their side effects, their shape type can simply be $\mathsf{Unit} \to \mathsf{Unit}$. Their capture set, however, is a more intricate matter. The deferred closures clearly need to be impure, since (once again) they are invoked for side effects. However, they cannot simply be allowed to capture arbitrary capabilities, since those capabilities may no longer be in scope when the closure is invoked. For an example, consider the following snippet, making use of our hypothetical effect-handler-based `defer`.

```
def foo() = withDefer { h =>
  try {
    h.defer { throw new Exception() }
  } catch {
    case _: Exception => ...
  }
}
```

We first introduce handler for `defer` with the `withDefer` function and then enter a **`try`** block which allows throwing `Exception`-s. Inside the **`try`** block, we defer a block that throws an `Exception`. The deferred block will be executed when we exit the **`try`** block, at which point it will throw an unhandled exception.

What is it that really went wrong? The deferred block was allowed to capture a scoped capability. Deferred blocks should only be able to capture capabilities which are guaranteed to be in scope when the blocks are executed, i.e., capabilities which *outlive* the defer scope. We can understand the problem by using the effect handler $\mathsf{CF}_{<:}$ extension to formally represent `withDefer` with the following term.

$$\begin{aligned}
\mathsf{withDefer} \triangleq\ & \Lambda[X <: \top \,\hat{}\, \{\mathbf{cap}\}]\, \Lambda[R <: \top \,\hat{}\, \{\mathbf{cap}\}] \\
& \quad \lambda(\mathit{thunk} : (\forall (\mathit{defer} : \mathsf{Eff}[(\mathsf{Unit} \to \mathsf{Unit}) \,\hat{}\, \{X\}, \mathsf{Unit}] \,\hat{}\, \{\mathbf{cap}\})\, R) \,\hat{}\, \{\mathbf{cap}\}) \\
& \quad\quad \mathbf{handle}\, x : \mathsf{Eff}[(\mathsf{Unit} \to \mathsf{Unit}) \,\hat{}\, \{X\}, \mathsf{Unit}] = \\
& \quad\quad\quad \lambda\left(f\, k\right)\, \mathbf{let}\, z = k\,()\ \mathbf{in}\ (f\,(); z) \\
& \quad\quad \mathbf{in}\ \mathit{thunk}\, x
\end{aligned}$$

First, let's take a close look at what this term does. The *thunk* argument to $\mathsf{withDefer}$ is the scope within which we can defer closures. The scope returns a value of arbitrary type R and receives *defer* as an argument: a capability of type $\mathsf{Eff}[(\mathsf{Unit} \to \mathsf{Unit}) \,\hat{}\, \{X\}, \mathsf{Unit}] \,\hat{}\, \{\mathbf{cap}\}$. Invoking this capability with a closure of type $(\mathsf{Unit} \to \mathsf{Unit}) \,\hat{}\, \{X\}$ defers the closure until the end of $\mathsf{withDefer}$. The capability is backed by a straightforward handler form $\lambda\left(f\, k\right)\, \mathbf{let}\, z = k\,()\ \mathbf{in}\ (f\,(); z)$. When the handler is invoked, it first runs k, the continuation of the defer scope, then it runs f, the deferred closure, and finally it returns the result of running k.

What is really interesting about this definition is that the type of the deferred closure, $(\mathsf{Unit} \to \mathsf{Unit}) \,\hat{}\, \{X\}$, disallows it from capturing a local capability! The type which would enable it do so would instead be $(\mathsf{Unit} \to \mathsf{Unit}) \,\hat{}\, \{\mathbf{cap}\}$, but this type is, as expected, disallowed by rule (HANDLE), since neither type argument to Eff can capture $\mathbf{cap}$. Instead, the capture set of the closure is only allowed to be $\{X\}$, where X is a type parameter of $\mathsf{withDefer}$. Recall that the $\mathsf{CF}_{<:}$ (T-APP) rule prevents type variables from being instantiated with a type capturing $\mathbf{cap}$. As I have argued in Section 2.1.5, this restriction ensures that all the capabilities captured by the instantiation of X must be in scope and available for the entire duration of $\mathsf{withDefer}$. Note also that X only appears in capture position, effectively making it a capture set parameter. To sum up, the base rules of $\mathsf{CF}_{<:}$ allow $\mathsf{withDefer}$ to be parameterized with a set of capabilities which are guaranteed to outlive it!

To conclude, the `defer` statement is a particular example of a facility which takes a closure and runs it at a later point, possibly in an outer lexical scope. Another example of such a facility is scheduling closures to run at an unspecified point. Properly expressing such facilities inside the language is possible only if the type system allows abstracting over a set of capabilities which are guaranteed to be in scope while the facility is accessible, and $\mathsf{CF}_{<:}$ impure type variables are one way to enable doing so.

5.2 Abstracting Over Arguments

Next, I present an issue that was discovered when experimenting with the implementation. The issue revolves around `PartialFunction`, a special type from the Scala standard library which can be described as a `Function` that knows where it is defined. `PartialFunction` is a trait defined by two abstract methods, `apply` and `isDefinedAt`.

```
trait PartialFunction[-A, +B] extends (A -> B):
  def apply(arg: A): B
  def isDefinedAt(arg: A): Boolean
```

The Scala compiler has support for `PartialFunction` literals: the following two definitions are equivalent, i.e., `head1` is elaborated into `head2`.

```
val head1: PartialFunction[List[Int], Int] =
  { case i :: is => i }

val head2 =
  class headImpl extends PartialFunction[List[Int], Int]:
    def apply(arg: List[Int]): Int =
      case i :: is => i

    def isDefinedAt(arg: List[Int]): Boolean =
      case i :: is => true
      case _ => false

  new headImpl()
```

Attempt to use `PartialFunction`-s together the Capture Tracking extension to Scala will result in rather puzzling errors. One way to encounter such errors would be to define a `PartialFunction` which destructures a `LazyList`. Since a `LazyList` builds itself lazily, it may capture capabilities which are necessary to calculate its elements, so our `PartialFunction` should range over `LazyList`-s capturing arbitrary capabilities. Such a definition might look as follows.

```
val headLL1: PartialFunction[LazyList[Int]^, Int] =
  { case n #:: ns => n }
```

However, the implementation will reject this snippet, informing us that type arguments (in this particular example, `LazyList[Int]^`) cannot be instantiated with a type capturing **`cap`**.

If we try to restrict ourselves to `PartialFunction`-s whose arguments have a concrete capture set, we will still encounter issues. In the following example, we have a file `f` and we intend to load all the lines from this file into a `LazyList`; such a `LazyList` has to capture `f`.

```
  val f: File^ = ...
  def lines(f: File^): LazyList[String]^{f} = ...

  val head: PartialFunction[LazyList[String]^{f}, String] =
    { case str #:: strs => str }
```

Even with this restriction in place, we will still get a compiler error.

To understand where the errors are coming from, we need to desugar the `PartialFunction` literals into classes, and we need to make box and unbox operations explicit.

```
val file: File^ = ...
val head2: PartialFunction[□ LazyList[String]^{f}, String]^{} =
  class headImpl extends PartialFunction[
    □ LazyList[String]^{f},
    String
  ] {
    def apply(arg: □ LazyList[String]^{f}) =
      ({f} ⟜ arg) match {
        case str #:: strs => str
      }

    def isDefinedAt(arg: □ LazyList[String]^{f}) =
      ({f} ⟜ arg) match {
        case str #:: strs => true
        case _ => false
      }
  }
  (new headImpl() :
    PartialFunction[□ LazyList[String]^{f}, String]^{f})
```

Now we can see that the type argument to `PartialFunction` is (and must be) boxed, which requires the bodies of `apply` and `isDefinedAt` to unbox the method arguments. Hence, any instance of `headImpl` will be charged with the capture set `{f}`, making the definition of `head2` ill-typed due to a mismatch between the highlighted capture sets.

The issue can be concisely stated as follows: given a class `C` which abstracts over the type of an argument, like `PartialFunction` does,[1] instances of `C` receive unexpected capture sets, ultimately due to $\mathsf{CC}_{<:\square}$ requiring all capturing type arguments to be boxed. The issue simply does not exist in $\mathsf{CF}_{<:}$, since its type variables are not boxed.

[1] Notably, the same issue is exhibited by all Single Abstract Method (SAM) types in Scala.

5.3 Mutable State

Finally, I present an issue which arises when working with mutable state.

When working with capture-polymorphic definitions using Capture Tracking, for the most part we can rely simply on the dependently-typed nature of the approach. Doing so is sufficient to express definitions such as a capture-polymorphic `map` and scales to expressing the Scala collections API (Section 4.5.3).

However, we can encounter issues when working with definitions that abstract over mutable state. As a formal illustration of these problems, consider the following term from the region extension to $\mathsf{CF}_{<:}$.

$$\begin{aligned}\mathsf{fn} \triangleq\ & \Lambda[A <: \top \,\hat{}\, \{\mathbf{cap}\}]\, \Lambda[B <: \top \,\hat{}\, \{\mathbf{cap}\}] \\ & \lambda(x : \mathsf{Ref}[\mathsf{Ref}[B] \,\hat{}\, \{A\}] \,\hat{}\, \{\mathbf{cap}\})\, \lambda(y : \mathsf{Ref}[B] \,\hat{}\, \{A\})\ t\end{aligned}$$

First, observe that fn can be applied to Ref-s whose contents have an arbitrary capture: it could be a singleton capture set $\{c\}$, or it could contain multiple variables, e.g., $\{c_1, c_2\}$, if the region of the inner Ref is not exactly known. Next, observe that t, the exact body of fn, is intentionally opaque. The fn function could perform a number of operations on x and y. It could write the contents of y to the contents of the inner cell of x, or the other way around, and it could write y to x directly.

This sort of flexibility is clearly useful when working with mutable data, even if it might only be necessary to express uncommon definitions. Between the $\mathsf{CF}_{<:}$ and $\mathsf{CC}_{<:\Box}$ approaches to type polymorphism, it is only supported by the former and not the latter. In particular, $\mathsf{CC}_{<:\Box}$ does not support "parametric capture polymorphism": it is not possible to abstract over an entire capture set by making it a parameter, like the $\mathsf{CF}_{<:}$ version of fn does with A. $\mathsf{CC}_{<:\Box}$ *does* support the following two alternative versions of fn.

$$\begin{aligned}\mathsf{fn}_1 \triangleq\ & \Lambda[A <: \top]\, \Lambda[B <: \Box\, \mathsf{Ref}[A] \,\hat{}\, \{\mathbf{cap}\}] \\ & \lambda(x : \mathsf{Ref}[B] \,\hat{}\, \{\mathbf{cap}\})\, \lambda(y : B)\ t\end{aligned}$$

$$\begin{aligned}\mathsf{fn}_2 \triangleq\ & \Lambda[B <: \top] \\ & \lambda(r : \top \,\hat{}\, \{\mathbf{cap}\})\, \lambda(x : \mathsf{Ref}[\mathsf{Ref}[B] \,\hat{}\, \{r\}] \,\hat{}\, \{\mathbf{cap}\})\, \lambda(y : \mathsf{Ref}[B] \,\hat{}\, \{r\})\ t\end{aligned}$$

However, both versions are less flexible than fn. In particular, fn_1 is disallowed from reading from and writing to y and the inner Ref of x because the bound of B is boxed, and fn_2 can only be applied when the capture of y and the contents of x is known to be a single capability.

While the term is specific to the particular $\mathsf{CF}_{<:}$ extension, the general situation is not. Mutable state cannot be allowed to capture **`cap`**, since doing so inherently allows accessing out-of-scope capabilities (see Section 2.1.5). And while the reason why the inner Ref has a capture in the first place is that Ref-s are allocated on Region-s in the $\mathsf{CF}_{<:}$ extension, similar scenarios are entirely possible if we define a class which has mutable state and needs to capture capabilities to carry out some computation, for instance if the class computes the initial mutable state lazily.

To conclude, being able to abstract over a capture set by giving it a name, i.e., parametric capture polymorphism, appears useful when working with mutable state. $\mathsf{CF}_{<:}$ gives us a way of extending Capture Tracking with parametric capture polymorphism by re-using type variable binders, without adding a new kind of binder to the language. Doing so is a natural extension of supporting impure type variables, since such variables need to appear in capture sets and at that point what we have is precisely parametric capture set polymorphism.

5.4 Conclusions

We have seen three classes of situations where the $\mathsf{CF}_{<:}$ approach to type polymorphism has advantages compared to $\mathsf{CC}_{<:\square}$. However, we want to keep the ergonomic data structure types afforded by $\mathsf{CC}_{<:\square}$ type polymorphism.. So, what are we to do?

One potential solution would be to integrate *both* forms of type polymorphism in a single system. Formally, there is nothing stopping us from doing so: we can have have a calculus with two different type abstraction forms and two different kinds of type variables.

The remaining question is that of language design: how to expose those two forms of polymorphism to the users? I propose that a natural way to do so might be to tie them to *variance of type parameters*. Observe that contravariant type parameters always range over types of *inputs*, by their very definition. One of the basic assumptions underlying Capture Tracking is that inputs are typically there to be used; this assumption is the basis of the argument that capability-style effect polymorphism is more natural (see the introduction to Chapter 2). By extension, the type variables which range over inputs *should not* prevent the inputs from being used.

Strictly tying the two forms of polymorphism to variance of type parameters would solve the problems with abstracting over argument types, but not the other two classes of problems we discussed. To solve them, we need proper parametric capture set polymorphism. I propose to allow explicitly specifying if a type parameter should be pure or impure, and to use the parameter's variance to select the default if the purity is unspecified. Doing so would solve all the three classes of problems we have seen and would expose a lightweight syntax for the common case where we want boxing covariant and invariant type parameters to enable ergonomic types and non-boxing contravariant type parameters to allow using inputs typed with a type parameter.

6 Gradual Compartmentalization

In the previous chapters we have seen a number of formal systems culminating in $\mathsf{CC}_{<:\square}$, which provided the formal foundations for the prototype Scala implementation of Capture Tracking. In this chapter my goal is to further argue for the practical applicability of Capture Tracking as a technique. To this end, I present the results of a collaboration with Adrien Ghosn, Clément Pit-Claudel, and Mathias Payer, which revolve around *gradual compartmentalization*: an approach to incrementally compartmentalizing a preexisting codebase via object capabilities tracked in types. The results are not yet published.

6.1 Introduction

Modern software development favors productivity over security. Application developers rely on diverse, unverified libraries written by unknown authors and downloaded off the Internet in order to extend their applications with basic functionality. In the extreme, modern application development becomes merely "gluing libraries together". This situation gave rise to *supply chain attacks*, a very dangerous attack vector. Finding a bug in a popular library, compromising a genuine one (e.g., by stealing its author's credentials) or publishing obfuscated malicious code can potentially grant access to hundreds of thousands of devices [Nikiforakis et al. 2012].

Modern software needs fine-grained compartmentalization, i.e., intra-process isolation. Ideally, application developers should be able to enforce the Principle of Least Authority (or Privilege) [Saltzer 1974; Melicher et al. 2017]: any software component's access to program and system resources should be limited to the minimum required for its correct operation.

Object capabilities are a particularly attractive approach to compartmentalization with a long history of research [Dennis and Van Horn 1966; Morris 1973; Rees 1996; Miller 2006; Melicher 2020]. The ocap discipline views all code in terms of objects and specifies that access to program and system resources is mediated via special objects: *capabilities*. Capabilities originate from the program's entrypoint; objects can only access a capability they received from another object, i.e., there is no ambient authority in the system. Packages are also viewed as objects, called *modules* [Melicher et al. 2017]. Since a module can only use capabilities it received from other objects, an application can control the authority of its components by controlling how capabilities are distributed.

Despite their clear advantages, ocap languages (e.g., E [Miller 2006], Newspeak [Bracha et al. 2010] or Wyvern [Melicher et al. 2017]) are not widely used in the industry. Arguably, this is precisely *because* they assume an application's code to have no ambient authority: existing applications were not written under such an assumption. If their developers want to reap the benefits of ocap, they face an extensive rewrite of their entire codebase, including the very libraries they introduced to the codebase to reduce their own labor.

In this chapter, I develop an approach for compartmentalizing an application which allows a *gradual* migration to object capabilities. Code at various levels of migration can coexist within a single application; this not only allows introducing the object capability discipline to the application component by component, but *also* allows extending an application with non-ocap components while still maintaining our desired security guarantees. I take inspiration from the idea of dynamically-enforced types from the gradual typing literature [Siek and Taha 2006; Wadler and Findler 2009; Wadler 2015], which allows values (equivalently, objects) to be dynamically typed, i.e., they can be used for any operation at the cost of potential runtime errors. The concept of dynamic enforcement is applied specifically to the *authority* of objects and not their entire types.

The key problem the approach solves is that until recently, it was unclear how to integrate existing non-ocap code with ocap code in a single application and still allow its components to be compartmentalized. The object capability discipline assumes no part of the system has ambient authority, while existing non-ocap code was written under no such assumption and may access arbitrary program and system resources. As a schematic example, in currently existing code a Log4J logger can simply be instantiated as:

```
(new log4j.Logger()).info("msg")
```

In contrast, an ocap version of the Logger class would need to be refactored to explicitly take the capabilities to access the filesystem (and the network [Chowdhury et al. 2022]) as arguments:

```
(new log4j.Logger(fs, net)).info("msg")
```

Ocap and non-ocap code seem to be fundamentally at odds. I alluded that mediating between them seems to inherently require *dynamically enforcing* the authority of non-ocap code. Such enforcement must be done *efficiently* enough to make the approach feasible in practice.

Recently, *Enclosures* [Ghosn et al. 2021] were proposed as an approach to compartmentalizing untrusted code which provides security guarantees even for foreign binaries thanks to relying on hardware support. An Enclosure restricts what program and system resources can be accessed in a given lexical scope; its restriction is expressed in terms of packages (and the memory associated with them) and system calls. Our key insight is that we can understand system calls as though they were method calls to a capability captured by the surrounding code, in addition to understanding mutable objects as capabilities. Doing so allows understanding existing non-ocap code as ocap code which was already initialized with some capabilities, *and* allows restricting the authority of such code at runtime with an Enclosure-like mechanism.

Furthermore, the type system is extended to verify that all foreign code had its authority restricted, either via dynamic checks or statically, with the type system itself. The extension uses *Capture Tracking* [Boruch-Gruszecki et al. 2023], a recently-published approach which augments types with *capture sets*, describing what capabilities each object has captured and therefore its authority. A particular advantage of Capture Tracking is its low annotation burden. Tracking the authority of objects in their types adds an *intermediate step* when migrating an application's component to object capabilities: the type system can be used to statically restrict the component's authority, without refactoring the component to take its desired capabilities as arguments.

Our contributions are as follows:

- *Gradual compartmentalization*, a hybrid approach which has the advantages of both dynamically-enforced and statically-verified compartmentalization and allows a gradual migration from one approach to the other.
- We discuss *Gradient*, a proof-of-concept gradual compartmentalization extension to the Scala language, in order to illustrate the key principles of our approach.
- We show the GradCC system to demonstrate Capture Tracking can be used to track authority of mutable objects even in presence of capture-unchecked terms.
- We validate that migrating existing Scala code to capture-checked, non-ocap Gradient code is practical by migrating Scala's standard XML library.

The rest of this section is organized as follows. First, I discuss additional background and motivation behind the approach (Section 6.2). Next, I present Gradient (Section 6.3). Then, I present the formal system (Section 6.4,Section 6.5) and finally I evaluate Gradient based on the experience of migrating a real-world Scala library and I discuss the feasibility of implementing Gradient (Section 6.6).

6.2 Background and Motivation

I distinguish and contrast between two salient ways of approaching compartmentalization: dynamic enforcement and static verification.

Dynamically-enforced compartmentalization is widely adopted in the industry. Examples include website sandboxing (e.g., Chromium tab isolation), containerization (e.g., Docker), systemd sandboxing, Linux application sandboxing (e.g., Snap, Flatpak), and mobile app permissions.

Dynamic mechanisms often operate at a coarse granularity, such as memory pages and processes. Compartmentalizing an existing application with such an approach is often challenging, requires heavy refactoring and incurs runtime costs. For instance, compartmentalizing an untrusted library with a process-based approach requires re-designing the application to run the untrusted code in a separate process and incurs the overhead of process switching and inter-process communication.

An important benefit of such low-level mechanisms is allowing *heterogenous* software written in any language, delivered as source code or as binaries. For instance, enclosures still provide security guarantees even in the presence of calls to foreign code which may forge arbitrary pointers.

Still, dynamic mechanisms naturally lead to runtime errors. Determining what policies to implement with such an approach is a matter of costly trial and error, since most software does not specify what permissions it needs. Overly broad policies weaken security; overly tight policies may cause runtime errors and prohibit expected functionality. Tellingly, Linux distributions do not agree on the systemd sandboxing restrictions placed on various services [Sandboxdb 2023] and the Java Security Manager was deprecated partly due to the "practically insurmountable challenge" [Java 2021] of determining appropriate security policies.

Statically-verified compartmentalization can be significantly more ergonomic, especially if it is integrated with a programming language. Such an approach inherently can assume code to be *homogenous*. It can tightly integrate enforcement of security policies with existing language constructs and types, dealing with objects rather than memory pages and system calls and scoping the restrictions to code blocks rather than to entire libraries. Such a mechanism can also statically verify if an application's components obey their intended system access restrictions, without incurring a performance cost *and* providing feedback quickly and reliably. Such feedback enables rapid development of security-conscious software and improves its maintainability: after any change, including a dependency update, security policies can be statically verified.

Gradual compartmentalization is a hybrid approach which lets the users adopt the best possible isolation strategy for each library:

1. Ocap code uses *object capabilities* as the principled compartmentalization mechanism.
2. Ocap code can interoperate with non-ocap code by leveraging *Capture Tracking* in order to track the authority of objects in types.
3. When all else fails, an Enclosures-inspired *runtime component* can dynamically enforce capability access restrictions, and Capture Tracking ensures the runtime component is used.

6.3 Gradient

The three key elements of gradual compartmentalization are object capabilities, tracking capabilities in types, and runtime authority enforcement. I present and discuss them based on Gradient, a proof-of-concept extension to the Scala programming language.

6.3.1 Object Capabilities

In Scala, similarly to most programming languages, ambient authority allows accessing system functionality simply by importing and using the appropriate packages. The object capability discipline, however, dictates that system functionality can be accessed only by calling methods on *capabilities*. As a result, Gradient code is organized in class-like units called *modules*.[1] Like classes, modules have constructors, which may take arguments. If the code within a module needs to use capabilities, the module needs to take such capabilities as constructor arguments. (In all our examples, the modules retain their constructor arguments as private fields.) For example, the following snippet shows an example Gradient program's entrypoint:

```
module Main(fs: Fs^, net: Net^):
  def main() =
    val logger = new Logger(fs)
    ... // do useful work
```

The program defines the `Main` module with a `main` method. The module's constructor takes two capability arguments: `fs` and `net`. They implement the `Fs` and `Net` interfaces, respectively, and are marked as capabilities by the hat `^` sign. The program starts by instantiating the `Main` module with the appropriate capabilities and calling its `main` method.

The `main` method itself begins by instantiating the Logger module, passing it the `fs` capability as an argument. The Logger module is defined as follows:

```
module Logger(fs: Fs^):
  def info(msg: String): Unit = ... // log the message
```

Organizing the code into ocap modules has some major benefits. First, it facilitates inspecting what system functionality may be accessed by an untrusted module. Gradient ensures that there is only a limited number of ways a module can gain *direct* access to a preexisting[2] capability; it can receive it as an argument (to a constructor or a method), receive it as a result of a method, or read it out of mutable state. For instance, to convince ourselves that `Logger` cannot access the network, we begin by checking that it does not receive a capability to do so

[1] Gradient modules are, if we disregard the gradual fragment, the same as Wyvern modules.

[2] Note that ocap allows objects, including modules, to create *some* capabilities out of nowhere. For instance, mutable objects in Wyvern are capabilities, and they can be created without access to any capability. Likewise, the extensions to $\mathsf{CF}_{<:}$ and $\mathsf{CC}_{<:\square}$ allow pure code which creates local effect capabilities. Perhaps one way to characterise such capabilities is by saying that creating them does not grant any additional authority over preexisting capabilities.

as a constructor argument. By further inspecting its API, we see it cannot receive this capability as a method argument either, and so it cannot access the network as desired.

Second, modules allow easily *attenuating* [Miller 2006] the authority gained through capabilities. Since a capability is just an object, we can create a wrapper capability around it and inspect every method call and its arguments to decide if it should be allowed; in a sense, doing so injects bespoke filters between a capability and its calling context. For instance, the following snippet schematically shows how `Main` can restrict `Logger` so that it can only access files in the "/var/log" directory.

```
module Main(fs: Fs^, net: Net^):
  def main() =
    val wfs = new Fs:
      def open(path: Path): FileHandle =
        if path.isRootedIn("/var/log") then fs.open(path)
        else throw IllegalArgumentException()

    val logger = new Logger(wfs)
    ... // do useful work
```

Interestingly, `Logger` itself is a capability that attenuates filesystem access granted by `fs`: it allows accessing the filesystem only to perform limited logging-related operations.

Strictly observing ocap allows the program's code to be naturally compartmentalized: security policies can be expressed by controlling the capabilities received by a module and attenuating their authority. For example, if the Log4j library [Chowdhury et al. 2022; Hiesgen et al. 2022] was implemented as an ocap module, all program using it would know it may access the network, since the Log4j module would require the `net` capability as an argument. Furthermore, if the library initially never accessed the network and only tried to do so after a (potentially malicious) update, programs attempting to use the new version of the library would not compile until their code was intentionally modified to grant the library additional capabilities.

There is one remaining piece of the puzzle: mutable state. If two modules share mutable state, they can communicate. They can exchange capabilities between each other, defeating compartmentalization attempts. More subtly, they can exchange information to influence their behaviour and make the recipient use their capabilities in a particular way, potentially leading to issues such as a confused deputy attack [Hardy 1988]. Ocap code is inherently more resistant to such attacks [Rajani et al. 2016]; in particular, ocap forbids global mutable variables: two modules can only communicate through mutable state if they both share access to the same mutable object. Gradient goes a step further and tracks all mutable objects in its type system, making it easier to verify that two modules cannot communicate. To explain this, I first discuss how Gradient employs Capture Tracking to track capabilities in types.

6.3.2 Capture Tracking

Gradient depends on the experimental Scala implementation of Capture Tracking to track capabilities in types; doing so facilitates reasoning about module compartmentalization.

Tracking capabilities

Capture Tracking makes capabilities visible in Gradient types, making it easier to verify if a module can potentially gain access to a capability. Consider the following variant signature of `Logger`:

```
module Logger(fs: Fs^):
  def info(lvl: Level, msg: String): Unit
```

To verify that an instance of `Logger` cannot gain access to a capability other than `fs`, we need to check it cannot receive it as a method argument. Normally, we would need to inspect the implementations of all the parameter types (i.e., both `Level` and `String` in our example) and verify that they cannot store a capability. Capture Tracking simplifies this, since it allows we can instead inspect their capture sets alone. In our example, both `lvl` and `msg` are untracked (their types have empty capture sets), meaning that `Logger` cannot receive a capability reference as an argument.

In particular, note that since Gradient treats all mutable objects as capabilities, the above signature also lets us know that both `lvl` and `msg` are deeply immutable, i.e., they are themselves immutable and do not grant access to (have not transitively captured) any mutable object.

Borrow safety

In some cases we may want to restrict a module from retaining a capability. Consider the following scenario:

```
module OCR():
  def update_models(net: Net^): Unit =
    ... // download updated models
  def ocr_pdf(stream: InputStream^): String =
    ... // OCR the contents of the stream

module Main(fs: Fs^, net: Net^):
  def main() =
    val ocr = new OCR()
    ocr.update_models(net)
    ... // do useful work involving OCR
```

`OCR`, a module with code for Optical Character Recognition defines `update_models`, a method for downloading an updated version of its internal models from the network. The same module defines `ocr_pdf`, a method which OCRs a file. If it retained `net` after the call to `update_models` it could exfiltrate private information from the files it OCRs.

Gradient uses Capture Tracking to statically rule out such problems and ensure *borrow safety*: a capability can only be retrieved out of mutable state if it is derived from other, already available capabilities. In the above example, `OCR` cannot store `net` in its own state and retrieve it later: `net` is derived from **`cap`** and `OCR` did not (in fact, cannot) receive **`cap`** during its instantiation.

Note that *some* capabilities can be read from mutable state. As an example, we can loop over all files in a directory and keep the current file in a mutable variable:

```
def foo(fs: Fs^) =
  val iter : Iterator[File^{fs}] = fs.children(...)
  var file : File^{fs} = null
  while iter.has_next():
    file = iter.next()
    ... // do useful work involving file
```

6.3.3 Runtime-Assisted Graduality

Currently existing code uses system calls to access system features, while ocap code calls methods on capabilities to do the same. Crucially, we can think of system calls *as though* they were method calls to a particular capability. Such capabilities are the most basic capabilities available to the program: I will call them *devices*, following Miller [2006]. Ocap code *also* uses devices to access system features: any module can request them as arguments while the `Main` module is special and it can be initialized with devices by the runtime.

For instance, `fs` is a device for accessing the file system. Devices are singleton objects: `fs` is the only such device. While devices appear as common objects to the program, their methods are runtime primitives which invoke system calls under the hood. For instance, if the `Main` module requests the `fs` device, the `main` method can call `fs.open`. Under the hood, this invokes the open system call. On the other hand, non-ocap code can invoke the same system call directly, which can be treated as calling a method on the `fs` device.

Still, how does non-ocap code receive the capabilities it needs? Non-ocap code in Gradient is organized in *packages*, the same as Scala code. Packages can be treated as objects which are *pre-initialized* by the runtime with the capabilities they need, even before the `Main` module is initialized and the `main` method is called. In addition, a Gradient package may optionally be *capture-unchecked*, which allows using any existing Scala package in Gradient with no migration cost, but at the cost of relying on a runtime component to ensure the package does not exceed its authority.

When migrating a preexisting Scala package to ocap, the first step is to make it capture-checked. Since the code in the package can still be written as though it had ambient authority, this first step in many cases should only require adding capture signatures to existing code, without needing to refactor it (see Section 6.6). The package can later be refactored into a module by rewriting the code so that it accepts the resources it needs as arguments from code outside the module; doing so allows the users of the module to attenuate the authority they grant to the module, as described in Section 6.3.1. Such an architecture resembles the "dependency injection" design pattern and arguably is a good software engineering practice [Miller 2006].

Capture-checked packages

The following example uses a capture-checked `Logger` *package*.

```
module Main(device fs, device net, package Logger):
  def main() =
    Logger.info("Starting...")
    ... // do useful work and log it
```

There is a number of differences between this version of `Main` and the one presented in Section 6.3.1. First, its constructor now requires special arguments, as signified by the **device** and **package** keywords and the lack of type ascriptions. The keywords signify that `fs` and `net` must be the appropriate *devices*, as opposed to arbitrary objects implementing the `Fs` and `Net` interfaces. Similarly, `Logger` must be the object representing the `Logger` package: it needs to be explicitly requested similarly to a device. Since there's only one possible instance of each of `fs`, `net` and `Logger` in the entire program, `Main` can request them by name without specifying their type, much like import statements do not require a type.

The `main` method can call `Logger.info`, presumably accesing the filesystem. This does not mean ocap is compromised: `Logger` was pre-instantiated with capabilities before `Main` was instantiated. The following example schematically illustrates how `Logger` is defined.

```
package Logger:
  def info(msg: String): Unit =
    fs.open("...").write(msg)
```

In the source, `Logger.info` can directly access `fs`, not unlike how existing Scala packages can access the filesystem by calling appropriate APIs. However, Gradient interprets this definition differently from baseline Scala: the **`package`** statement logically defines a first-class object available in the global lexical scope. Non-ocap code in other packages can directly use it, while ocap modules need to explicitly require it as an argument.

Enforcing security policies on non-ocap packages presents some difficulties, since they are initialized with capabilities even before `Main`. The ability to attenuate capability access is lost; Capture Tracking, however, still tracks the capabilities captured by `Logger` in its type.

Gradient allows using this information to check at compile time what capabilities `Logger` may access, using a **`restricted`** block.

```
module Main(device fs, device net, package Logger):
  def main() =
    restricted[{fs}] { Logger.info("Starting...") }
  ... // do useful work and log it
```

A **`restricted`** block is the equivalent type ascription for capability access: the ascribed block (`{ Logger.info("Starting...") }` in the example) can only access capabilities from the ascription (`{fs}` in the example, note it is accessed indirectly by `Logger`). For convenience, these ascriptions can be collected into a module.

```
module SafeLogger(device fs, package Logger):
  def info(msg: String): Unit =
    restricted[{fs}] { Logger.info(msg) }
```

In either case, thanks to Capture Tracking we re-gain the desired security guarantees. Our program statically checks if `Logger` accesses only the devices we permit it to access.

We have seen that we can integrate non-ocap and ocap code without compromising the compartmentalization guarantees or essential aspects of object capabilities. Still, we have assumed that the `Logger` package has capture signatures, i.e., it is capture-checked. Naturally, this is not the case for arbitrary existing Scala code: assigning it capture signatures involves a degree of manual work and in edge cases may require refactoring the code.

Capture-unchecked packages

As the name suggests, the signatures of a capture-unchecked package do not mention capture sets. This means we cannot rely on the **`restricted`** block to restrict the authority of such a package. Instead, the **`enclosed`** block can be used to dynamically restrict access to devices using a *runtime component*. The type system ensures that all code from capture-unchecked packages is run in an **`enclosed`** block. Syntactically, using a capture-unchecked package is similar to previous examples:

```
#package Logger:
  def info(msg: String): Unit = fs.open("...").write(msg)

module Main(device fs, device net, #package Logger):
  def main() =
    enclosed[{fs}] { Logger.info("Starting...") }
    ... // do useful work and log it
```

An **`enclosed`** block operates at a lower granularity than a **`restricted`** block: its restriction can only mention devices and *regions*. All modules have an associated memory region, and all mutable objects are at creation associated with such a region. The runtime component can efficiently check that only the specified devices and regions are accessed. If an `enclosed` block exceeds its restriction due to capture-unchecked code, its execution will be aborted and an exception will occur; capture-checked code can still be verified statically.

6.4 Base Formalism

This section, as well as the following one, present the formal foundations for Gradient, split into two fragments. I start by presenting ModCC, the fragment which accounts for capture-checked Gradient programs and their features: modules, packages and mutable state. Later I present GradCC, which allows formally representing capture-unchecked code.

Recall that the notation $\overline{E_i}^i$ denotes a *syntactic repetition* of a non-negative number of syntax forms E_i. If the individual forms never occur alone, the index can be omitted, as in $\overline{E}$. Furthermore, the notation $\overline{E}^{0..1}$ denotes an *optional* occurence of E.

6.4.1 Syntax

I begin by giving an overview of ModCC and its syntax; we discuss how ModCC can formally represent Gradient programs at the end of this subsection.

ModCC is based on $\mathsf{CC}_{<:\Box}$, since Gradient depends on the Scala implementation of Capture Tracking, itself grounded in the $\mathsf{CC}_{<:\Box}$ formalism.Recall that $\mathsf{CC}_{<:\Box}$ is a capture-tracked version of System $\mathsf{F}_{<:}$ with boxes and MNF terms. In turn, ModCC is a version of $\mathsf{CC}_{<:\Box}$ without type polymorphism and with modules, mutable state, and paths instead of variables.

Paths. ModCC allows selecting module fields with *paths* p. A path $x.\overline{f}$ is a *root variable* x followed by a zero or more *field selections* $\overline{.f}$. ModCC paths effectively replace $\mathsf{CC}_{<:\Box}$ variables: operands in terms and capture set elements both are paths, where they were variables in $\mathsf{CC}_{<:\Box}$. Additionally, an ModCC path can be looked up in a typing context $\Gamma(p) \to T$ (Figure 6.2), much like a $\mathsf{CC}_{<:\Box}$ variable could be looked up with $\Gamma \ni x : T$.

Dependent types. ModCC types may refer to a term variable if it occurs in a capture set; accordingly, function types have the form $\forall(x : T_1)\, T_2$ to allow x to occur in T_2.

Capture Tracking. ModCC types are partitioned into *shape types* S and regular types T. Syntactically, the latter are *capturing types* $S \,\hat{}\, C$, where the *capture set* C is a set of paths. We freely use shape types as regular types, assuming that $S \equiv S \,\hat{}\, \{\}$. Shape types comprise boxed types and the usual types of values.

Boxes. ModCC inherits box forms $\Box\, p$ and unbox forms $C \multimap p$ from $\mathsf{CC}_{<:\Box}$. Boxing a capability *temporarily* prevents it from counting as captured by the surrounding term; its type also becomes a pure *boxed type* $\Box\, T$. In order to use such a capability, first it needs to be *unboxed*

Variable	$x, y, z, \mathbf{cap}$		
Field	$f, \mathbf{reg}$		
Path	p, q	$::=$	$x.\overline{f}$
Value	v, w	$::=$	$\Box p \mid \lambda(x : T)\, t \mid \{\overline{f = p}\} \mid ()$
Term	t, u	$::=$	$p \mid v \mid p\, p \mid C \multimapinv p \mid \mathbf{let}\, x = t\, \mathbf{in}\, t \mid \mathbf{region}$
			$\mid\ !p \mid p := p \mid p.\mathbf{ref}\, p \mid \mathbf{mod}(p)\, \{\overline{f = p}\}$
Shape Type	S, R	$::=$	$\top \mid \forall(x : T)\, T \mid \Box T \mid \mathsf{Unit}$
			$\mid \mathsf{Ref}[S] \mid \mathsf{Reg} \mid \overline{\mu x}^{0..1}\, \{\overline{f : T}\}$
Type	T, U	$::=$	$S \wedge C$
Capture Set	C, D	$::=$	$\{\overline{p}\}$
Typing Context	Γ, Δ	$::=$	$\varnothing \mid \Gamma, x : T \quad \mathbf{if}\ x \neq \mathbf{cap}$

Figure 6.1: ModCC syntax. Highlighted forms are new compared to CC$_{<:\Box}$.

Context lookup $\qquad \Gamma \vdash p\, \mathbf{bd} \iff \exists T.\ \Gamma(p) \to T \qquad \boxed{\Gamma(p) \to T}$

$$\frac{\Gamma \ni x : S \wedge C}{\Gamma(x) \to S \wedge C} \qquad\qquad \frac{\Gamma(p) \to \overline{\mu y}^{y}\, \{\overline{f_i : T_{f_i}}^{i}\} \wedge C}{\Gamma(p.f) \to [\overline{y := x}^{y}]\, T_f}$$

Figure 6.2: Context lookup rules.

$C \multimapinv p$, which can only be done at the cost of counting the capabilities in C as captured by the surrounding term.

Recall that both box and unbox operations are statically inferred by the compiler; they are specific to the formal system and not a feature of the surface language.

Boxes allow formally representing the interaction between capabilities and mutable state (6.3.2): since the contents of mutable references $\mathsf{Ref}[S]$ must be pure, a capability can only be stored if it is boxed. An object which reads a capability out of mutable state has to unbox it before it can be used, which can only be done if the object's capture set accounts for the obtained capabilities.

References and regions. ModCC features *mutable references* which can be written to and read from, which are always associated with a *region*. (ModCC regions are never deallocated, unlike the regions discussed in Section 3.3.4.) A reference is created with the $p.\mathbf{ref}\ q$ form, where q is the initial value of the reference and p is a *region capability*, itself created with the **region** form. Importantly, no capability is necessary to create a region, which allows creating regions and

Definition 6.1 (Capture Set Operations)**.** *We define the following path-aware set operations.*

$$C \ominus x \triangleq \{\, y.\overline{f} \in C \mid y \neq x \,\} \qquad x \propto C \triangleq x \in \{\, y \mid y.\overline{f} \in C \,\}$$

Definition 6.2. *The captured paths are given by the* cv *function, defined as follows.*

$$\begin{array}{lll}
\mathrm{cv}(p) & \triangleq & \{p\} \\
\mathrm{cv}(\Box\, p) & \triangleq & \{\} \\
\mathrm{cv}(\lambda(x : U)\, t) & \triangleq & \mathrm{cv}(t) \ominus x \\
\mathrm{cv}(\{\overline{f_i = p_i}^{\,i}\}) & \triangleq & \{\overline{p_i}^{\,i}\} \\
\mathrm{cv}(\textbf{\textit{let}}\, x = v\; \textbf{\textit{in}}\, t) & \triangleq & \mathrm{cv}(t) \qquad\qquad \textbf{\textit{if}}\, x \not\propto \mathrm{cv}(t) \\
\mathrm{cv}(\textbf{\textit{let}}\, x = u\; \textbf{\textit{in}}\, t) & \triangleq & \mathrm{cv}(u) \cup \mathrm{cv}(t) \ominus x \\
\mathrm{cv}(p\, q) & \triangleq & \{p, q\} \\
\mathrm{cv}(C \multimapinv p) & \triangleq & C \cup \{p\} \\
\mathrm{cv}(p.\textbf{\textit{ref}}\, q) & \triangleq & \{p, q\} \\
\mathrm{cv}(\textbf{\textit{mod}}(p)\, \{\overline{f_i = q_i}^{\,i}\}) & \triangleq & \{p, \overline{q_i}^{\,i}\} \\
\mathrm{cv}(\textbf{\textit{region}}) & \triangleq & \{\}
\end{array}$$

Figure 6.3: The definition of cv.

using them to allocate local mutable state even inside untracked (pure) functions. References can be read with the $!p$ form and written with the $p := q$ form.

Records and modules. ModCC extends $\mathsf{CC}_{<:\Box}$ with records $\{\overline{f = p}\}$ and their usual semantics. In addition, ModCC also features *modules*: special records which can be created with the $\textbf{mod}(p)\,\{\overline{f = q}\}$ form. Doing so creates a record that packs together a region capability p with other values $\overline{q}$ (the bodies of fields $\overline{f}$); the region capability is stored in the special field **reg** and the field bodies may reference the packed region capability. Both records and modules are typed with the record type $\overline{\mu x}\,\{\overline{f : T}\}$, which allows an optional recursive qualifier. In a sense, ModCC modules are like a specialized version of ML modules [Mitchell and Harper 1988]: a Gradient module is always parameterized with a single region. We borrow the idea of modeling objects as records from DOT [Amin et al. 2016]; our record type features a recursive qualifier analogous to variable-recursive types from DOT. The qualifier is useful specifically for modules whose fields reference the region packed together with the module.

Captured Capabilities. In Section 6.3, we saw that Gradient uses capture sets to reason about capabilities captured by objects. To formally reason about captured capabilities, we use *paths* rooted in free variables of terms representing objects; such variables will be substituted with store locations which may contain capabilities. We define the cv function (Figure 6.3) to calculate such *captured paths*. Essentially, cv is a close cousin of fv which accounts for boxing and ANF:

- A boxed path $\Box\, p$ does not count as captured. Dually, for an unbox form $C \multimapinv p$ only the "key" C counts as captured.

- A let-bound variable, the v in $\mathbf{let}\, x = v\, \mathbf{in}\, t$, is only considered captured if it, or paths rooted in it, are captured by t.

Using paths instead of variables (i.e., defining $\mathrm{cv}(x.f) \triangleq \{x.f\}$) increases the precision of cv when dealing with records and modules. The following example shows how this affects ModCC typing:

$$\begin{aligned}&\mathsf{fn} : \forall(x : \{f_1 : \mathsf{Proc}\^\{\mathbf{cap}\}, f_2 : \mathsf{Proc}\^\{\mathbf{cap}\}\}\^\{\mathbf{cap}\})\ \{f_0 : \mathsf{Proc}\^\{x.f_1\}\}\^\{x.f_1\}\\ &\mathsf{fn} = \lambda(x : \{f_1 : \mathsf{Proc}\^\{\mathbf{cap}\}, f_2 : \mathsf{Proc}\^\{\mathbf{cap}\}\}\^\{\mathbf{cap}\})\ \{f_0 = \lambda(x : \mathsf{Unit})\ x.f_1\ ()\}\end{aligned}$$

The result of fn captures only $\{x.f_1\}$, i.e., a single field of x. As a consequence, if fn is called with an argument whose field f_1 is pure, the result of the call will be pure no matter what is captured by the other fields of the argument.

Gradient and ModCC. ModCC is intended to be the formal foundation underlying Gradient. Gradient modules can be translated to a formal ModCC term much like Scala classes can be translated to a DOT term [Amin et al. 2016; Martres 2023]. Concretely, a module corresponds to a Gradient function which formally represents the module's constructor: it takes the constructor's arguments, creates a fresh region for the module and creates the module itself, as illustrated in the following example.

```
module Logger(fs: Fs^):                let newLogger = λ(fs: Fs^)
  def log(msg: String): Unit = ...       let r = region in
                                         let _log = λ(msg: String) ... in
                                         mod(r) { log = _log }
                                       in

module Main(fs: Fs^, net: Net^):       let newMain = λ(fs: Fs^) λ(net: Net^)
  def main() =                           let r = region in
    val logger = new Logger(fs)          let _main = λ(u: Unit)
    ...                                    let logger = newLogger fs in ...
                                         in mod(r) { main = _main }
                                       in
                                       // initialize Main & run the program
                                       let main = newMain fs net in
                                       main.main ()
```

The example also shows that a Gradient program corresponds to a ModCC *term*. The Gradient program comprises contains module and package definitions which correspond to let-bound ModCC terms;[3] the body of the innermost let term initializes the packages (if there are any) and the `Main` module, and proceeds to run the program by calling the `main` method. We treat

[3]Technically, the translation approach demonstrated in the example does not allow for mutually-recursive modules. Possibly, such modules could be supported by a different translation scheme where modules are stored in and read from mutable state (simulating lazy let bindings, which do allow mutually-recursive modules). Alternatively, ModCC could be extended with record/module forms with field initializers, as in pDOT [Rapoport and Lhoták 2019]. Doing so would allow definining mutally-recursive modules as fields of a single record.

Gradient devices such as `fs` and `net` (and their types) as extensions to the base formalism; we do not privilege any particular device by baking it into the formal system.

6.4.2 Subcapturing

Figure 6.4 shows the subcapturing rules of ModCC. Subcapturing consistently uses paths instead of variables; accordingly, rule (SC-PATH) uses path lookup and replaces rule (SC-VAR) from $\mathsf{CC}_{<:\square}$, which only looked up variables. Rules (SC-ELEM) and (SC-SET) are directly inherited from $\mathsf{CC}_{<:\square}$. In addition, rule (SC-MEM) allows relating a module's field to the module itself. For simplicity, ModCC subcapturing features a separate transitivity rule (SC-TRANS), while transitivity in $\mathsf{CC}_{<:\square}$ was inlined into premises of subcapturing rules and therefore an admissible property.

6.4.3 Subtyping

Nearly all subtyping rules of ModCC are inherited from $\mathsf{CC}_{<:\square}$ (Figure 6.4). Like in $\mathsf{CC}_{<:\square}$, rule (CAPT) connects subtyping to subcapturing. Rule (REC) is the standard breadth-and-width rule for subtyping records. Since reference types are invariant, they do not have a dedicated subtyping rule and can only be compared with (REFL). Like in the DOT family of systems, recursive types do not participate in subtyping; instead, they can be eliminated and introduced in typing.

6.4.4 Typing

Figure 6.4 presents our typing rules. We inherit all the typing rules of $\mathsf{CC}_{<:\square}$, with small adjustments to account for paths replacing variables: the (VAR) variable typing rule from $\mathsf{CC}_{<:\square}$ is replaced with the path typing rule (PATH), while the $\mathsf{CC}_{<:\square}$ rules (BOX) and (UNBOX) use path lookup to ensure that the typing context binds all capture set elements. Rules (PACK) and (UNPACK) allow packing and unpacking recursive qualifiers on module types. Rule (ABS) types term abstractions; it uses the cv of the abstraction term as the assigned capture set. Rule (APP) types term applications $p\,q$. Since the result of a function type may depend on its parameter, (APP) replaces such parameter occurences with the concrete argument applied to the abstraction. Rules (LET), (SUB) are standard.

Rules (REGION) and (REF) type region and reference creation forms, respectively. The capture set assigned to a reference is the region capability used to create it. Rule (REF) ensures that only pure, untracked objects can be stored in references. As explained in Section 6.4.1, this forces tracked objects to be boxed before they can be stored in mutable state. If a tracked object is read out of mutable state, it needs to be unboxed before it can be accessed. Doing so adds the capabilities used to unbox the object to the cv of the unboxing term, which guarantees borrow safety (6.3.2).

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

SC-PATH
$$\frac{\Gamma(p) \to S\,\hat{}\,C}{\Gamma \vdash \{p\} <: C}$$

SC-ELEM
$$\frac{p \in D}{\Gamma \vdash \{p\} <: D}$$

SC-MEM
$$\frac{\Gamma \vdash p.f\ \mathbf{bd}}{\Gamma \vdash \{p.f\} <: \{p\}}$$

SC-SET
$$\frac{\overline{\Gamma \vdash \{p_i\} <: D}^{\,i}}{\Gamma \vdash \{\overline{p_i}^{\,i}\} <: D}$$

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

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

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

TOP
$$\Gamma \vdash S <: \top$$

REFL
$$\Gamma \vdash T <: T$$

TRANS
$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3}$$

BOXED
$$\frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash \Box\, T_1 <: \Box\, T_2}$$

FUN
$$\frac{\Gamma \vdash U_2 <: U_1 \qquad \Gamma, x : U_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x : U_1)\, T_1 <: \forall(x : U_2)\, T_2}$$

REC
$$\frac{\overline{\Gamma \vdash U_{f_j} <: T_{f_j}}^{\,j}}{\Gamma \vdash \{\overline{f_i : U_{f_i}}^{\,i}\} <: \{\overline{f_j : T_{f_j}}^{\,j}\}}$$

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

UNIT
$$\Gamma \vdash () : \mathsf{Unit}$$

PATH
$$\frac{\Gamma(p) \to S\,\hat{}\,C}{\Gamma \vdash p : S\,\hat{}\,\{p\}}$$

UNPACK
$$\frac{\Gamma \vdash p : \mu x\{\overline{f : T}\}\,\hat{}\,C}{\Gamma \vdash p : ([x := p]\{\overline{f : T}\})\,\hat{}\,C}$$

PACK
$$\frac{\Gamma \vdash p : ([x := p]\{\overline{f : T}\})\,\hat{}\,C}{\Gamma \vdash p : \mu x\{\overline{f : T}\}\,\hat{}\,C}$$

ABS
$$\frac{\Gamma, x : U \vdash t : T \qquad \Gamma \vdash U\ \mathbf{wf}}{\Gamma \vdash \lambda(x : U)\, t : (\forall(x : U)\, T)\,\hat{}\,(\mathrm{cv}(t) \ominus x)}$$

APP
$$\frac{\Gamma \vdash p : (\forall(x : U)\, T)\,\hat{}\,C \qquad \Gamma \vdash q : U}{\Gamma \vdash p\, q : [z := q]\, T}$$

LET
$$\frac{\Gamma \vdash u : T \qquad \Gamma, x : T \vdash t : U \qquad x \not\in \mathrm{fv}(U)}{\Gamma \vdash \mathbf{let}\, x = u\, \mathbf{in}\, t : U}$$

SUB
$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T <: U \qquad \Gamma \vdash U\ \mathbf{wf}}{\Gamma \vdash t : U}$$

BOX
$$\frac{\Gamma \vdash p : S\,\hat{}\,C \qquad \overline{\Gamma \vdash q\ \mathbf{bd}}^{\,q \in C}}{\Gamma \vdash \Box\, p : \Box\, S\,\hat{}\,C}$$

UNBOX
$$\frac{\Gamma \vdash p : \Box\, S\,\hat{}\,C \qquad \overline{\Gamma \vdash q\ \mathbf{bd}}^{\,q \in C}}{\Gamma \vdash C \multimap p : S\,\hat{}\,C}$$

REGION
$$\Gamma \vdash \mathbf{region} : \mathrm{Reg}\,\hat{}\,\{\mathbf{cap}\}$$

REF
$$\frac{\Gamma \vdash p : \mathrm{Reg}\,\hat{}\,\{\mathbf{cap}\} \qquad \Gamma \vdash q : S}{\Gamma \vdash p.\mathbf{ref}\, q : \mathrm{Ref}[S]\,\hat{}\,\{p\}}$$

READ
$$\frac{\Gamma \vdash p : \mathrm{Ref}[S]\,\hat{}\,\{\mathbf{cap}\}}{\Gamma \vdash\, !p : S}$$

WRITE
$$\frac{\Gamma \vdash p : \mathrm{Ref}[S]\,\hat{}\,\{\mathbf{cap}\} \qquad \Gamma \vdash q : S}{\Gamma \vdash p := q : \mathsf{Unit}}$$

RECORD
$$\frac{\overline{\Gamma \vdash p_i : S_i\,\hat{}\,C_i}^{\,i}}{\Gamma \vdash \{\overline{f_i = p_i}^{\,i}\} : \{\overline{f_i : S_i\,\hat{}\,C_i}^{\,i}\}\,\hat{}\,(\textstyle\bigcup_i C_i)}$$

MODULE
$$\frac{\Gamma \vdash q : \mathrm{Reg}\,\hat{}\,\{\mathbf{cap}\} \qquad \overline{\Gamma \vdash p_i : U_i}^{\,i} \qquad \overline{T_i = [q := x.\mathbf{reg}]\, U_i}^{\,i}}{\Gamma \vdash \mathbf{mod}(q)\, \{\overline{f_i = p_i}^{\,i}\} : \mu x\{\mathbf{reg} : \mathrm{Reg}\,\hat{}\,\{\mathbf{cap}\}, \overline{f_i : T_i}^{\,i}\}\,\hat{}\,\{\mathbf{cap}\}}$$

Figure 6.4: ModCC static rules. Highlighted rules and premises are new or changed (resp.) compared to $\mathrm{CC}_{<:\Box}$.

Rules (READ) and (WRITE) type read and write forms. Finally, rule (RECORD) is the standard record typing rule; the capture set of a record is the union of the field capture sets. Rule (MODULE) is a variant of (RECORD): a ModCC module is a record "packed" together with a region. For each field f_i, the rule requires the field's body p_i, to be typeable at some type U_i. However, in the entire module's type the type of each field f_i is instead $[q := x.\textbf{reg}]U_i$, where q is the region packed with the module and x is the DOT-like *recursive self-reference*. Finally, since a module packs a region into itself, the capture set of the entire module is simply $\{\textbf{cap}\}$.

6.4.5 Reduction

Figure 6.5 shows our reduction rules and runtime-specific forms. Unlike $\mathsf{CC}_{<:\Box}$, ModCC reduces store-term *configuration* pairs (σ, t). The term t is decomposed into an *evaluation context* η and a potential redex u. The rules are deterministic: at any point there is at most one applicable rule.

Reduction rules use l for store locations and r for paths rooted in locations. Rather than treating locations as a different grammatical category and defining additional typing rules, we make the simplifying assumption that they are variables. Stores σ comprise location-entry pairs $l \mapsto e$; an entry e is either a value, a region, a region-associated reference or a module.

Rules (APPLY), (TAPPLY), (OPEN), (RENAME) and (LIFT) are inherited from $\mathsf{CC}_{<:\Box}$. Rules (GET) and (SET) reduce mutable state reads and writes. Reference and module creation forms are reduced by rules (ALLOC) and (MALLOC). Because the fields of records and modules in the store always point to other paths and ultimately resolve to a location, runtime paths are effectively aliases for locations. Store lookup is aware of such aliases, e.g., given $\sigma = l_1 \mapsto \{f = l_2\}, l_2 \mapsto v$ we have $\sigma(l_1.f) = \sigma(l_2) = v$.

6.4.6 Metatheory

I show that ModCC is sound with the standard Progress and Preservation Theorems [Wright and Felleisen 1994]. The metatheory of ModCC is developed following the Barendregt convention: we only consider typing contexts where all variables are unique, i.e., for all contexts of the form $\Gamma, x : T$ we have $x \notin \mathrm{dom}(\Gamma)$.

As usual, a definition of store typing $\sigma \sim \Delta$ is necessary. (As a convention, we use Δ to refer to typing contexts related to stores.) It is defined in terms of store entry typing $\Delta \vdash l \mapsto e \sim \Delta$ as follows:

Definition 6.3. *We have* $\overline{l_i \mapsto e_i}^{\,i} \sim \Delta$ *if and only if:*

1. *We have both* $\overline{\Delta \vdash l_i \mapsto e_i \sim \Delta_i}^{\,i}$ *and* $\Delta = \overline{\Delta_i}^{\,i}$.
2. *If* e_i *is a record* $\{\overline{f_j = r_j}^{\,j}\}$ *or a module* $\textbf{\textit{mod}}(r')\,\{\overline{f_j = r_j}^{\,j}\}$,
   then for some T_i *we have* $\Delta = \Delta', l_i : T_i, \Delta''$ *and we have* $\overline{\Delta' \vdash r_j\ \textbf{\textit{bd}}}^{\,j}$.

Reduction $\boxed{(\sigma; t) \longrightarrow (\sigma; t)}$

$$\begin{array}{llll}
(\sigma;\eta[\, r\, r' \,]) & \longrightarrow\ (\sigma\,;\eta[\, [x := r']t \,]) & \textbf{if}\ \sigma(r) = \lambda(x:T)\, t & (\textsc{apply}) \\
(\sigma;\eta[\, C \multimapinv r \,]) & \longrightarrow\ (\sigma\,;\eta[\, r' \,]) & \textbf{if}\ \sigma(r) = \Box\, r' & (\textsc{open}) \\
(\sigma;\eta[\, !r \,]) & \longrightarrow\ (\sigma\,;\eta[\, v \,]) & \textbf{if}\ \sigma(r) = l \rhd \textbf{ref}\ v & (\textsc{get}) \\
(\sigma;\eta[\, r := r' \,]) & \longrightarrow\ (\sigma';\eta[\, () \,]) & \textbf{if}\ \sigma' = [r \mapsto \sigma(r')]\sigma & (\textsc{set}) \\
(\sigma;\eta[\, v \,]) & \longrightarrow\ (\sigma, l \mapsto v;\eta[\, l \,]) & \textbf{if}\ l\ \text{fresh} & (\textsc{lift}) \\
(\sigma;\eta[\, \textbf{let}\, x = r\, \textbf{in}\, t \,]) & \longrightarrow\ (\sigma\,;\eta[\, [x := r]t \,]) & & (\textsc{rename}) \\
(\sigma;\eta[\, r.\textbf{ref}\, r' \,]) & \longrightarrow\ (\sigma, l \mapsto e;\eta[\, l \,]) & & \\
 & \multicolumn{2}{l}{\textbf{if}\ \ l\ \text{fresh},\ \sigma(r) = \textbf{region}_{l'},\ e = l' \rhd \textbf{ref}\, \sigma(r')} & (\textsc{alloc}) \\
(\sigma;\eta[\, \textbf{mod}(r)\, \{\overline{f = r'}\} \,]) & \longrightarrow\ (\sigma, l \mapsto e;\eta[\, l \,]) & & \\
 & \multicolumn{2}{l}{\textbf{if}\ \ l\ \text{fresh},\ \sigma(r) = \textbf{region}_{l'},\ e = \textbf{mod}(l')\, \{\overline{f = r'}\}} & (\textsc{malloc})
\end{array}$$

$$\begin{array}{llll}
\textbf{Variable} & l, \ldots & & \\
\textbf{Store context} & \sigma & ::= & \overline{l \mapsto e} \\
\textbf{Eval context} & \eta & ::= & [\,] \mid \textbf{let}\, x = \eta\, \textbf{in}\, t \\
\textbf{Store entry} & e & ::= & v \mid \textbf{region}_l \mid l \rhd \textbf{ref}\, v \mid \textbf{mod}(l)\, \{\overline{f = r}\} \\
\textbf{Runtime path} & r & ::= & l \mid r.f
\end{array}$$

Figure 6.5: ModCC operational semantics.

The first condition connects store typing to store entry typing: the typing context of the former must be assembled out of fragments built by the latter. The second condition is a well-formedness criterion for stores: bodies of modules can only refer to paths bound *before* the module is bound. and likewise for records. Most of the store entry typing rules are the same as their corresponding typing rules ((UNIT), (BOX), (ABS), (REF)), e.g., if $\Delta \triangleq l_1 : \text{Reg}\,\hat{}\,\{\textbf{cap}\}, l_2 : \text{Ref}[\text{Unit}]\,\hat{}\,\{\textbf{cap}\}$ we have both $\Delta \vdash l_1 \mapsto \textbf{region}_{l_1} : \text{Reg}\,\hat{}\,\{\textbf{cap}\}$ and $\Delta \vdash l_2 \mapsto l_1 \rhd \textbf{ref}\ () \sim l_2 : \text{Ref}[\text{Unit}]\,\hat{}\,\{\textbf{cap}\}$.

The store typing rules for records and modules are slightly different than their corresponding typing rules, since they also add *path aliases* $p \equiv q$ to the output (Figure 6.6); the syntax of typing contexts is extended to allow such aliases. The primary reason the metatheory needs to be aware of path aliases is that during reduction, given a region packed with a module, a direct reference to the region must be equivalent to referencing it through the module. Furthermore, a module may refer to a region indirectly, through one or more fields of another module or a record. Hence, a record's or a module's fields should be equivalent to their bodies in general. Finally, all aliased paths must be bound to an equivalent type in the store-corresponding typing context. To ensure this is the case, both (ST-RECORD) and (ST-MODULE) use path lookup to assign types to the bodies of fields.

Path aliases are a minimalistic version of singleton types studied in pDOT [Rapoport and Lhoták 2019]; similarly, using path lookup instead of typing resembles the concept of strict typing used to prove the soundness of systems from the DOT family [Amin et al. 2014; Rapoport and Lhoták 2019; Boruch-Gruszecki et al. 2022]. Both are a proof device for establishing the

Store entry typing $\boxed{\Delta \vdash l \mapsto e \sim \Delta}$

ST-RECORD
$$\frac{\overline{\Delta(r_i) \to T_i}^{\,i}}{\Delta \vdash l \mapsto \{\overline{f_i = r_i}^{\,i}\} \sim l : \{\overline{f_i : T_i}^{\,i}\}, \overline{l.f_i \equiv r_i}^{\,i}}$$

ST-MODULE
$$\frac{\Delta \vdash l : \mathrm{Reg}\char`^\{\mathbf{cap}\} \qquad \overline{\Delta(r_i) \to U_i}^{\,i} \qquad \overline{T_i = [l := x.\mathbf{reg}]U_i}^{\,i}}{\Delta \vdash l_0 \mapsto \mathbf{mod}(l)\,\{\overline{f_i = r_i}^{\,i}\} \sim l_0 : \mu x\,\{\overline{f_i : T_i}^{\,i}\}, l \equiv l_0.\mathbf{reg}, \overline{l.f_i \equiv r_i}^{\,i}}$$

Figure 6.6: Some ModCC store entry typing rules. The full version is attached in the appendix.

soundness properties of ModCC, rather than a core feature of the system; they are presented as part of the metatheory and not of the formal system proper since an alternative proof which does not rely on them would be equally valid.

The Progress and Preservation Theorems are stated as follows.

Theorem 6.1 (Progress). *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$. Then either there exists r such that $t = r$, or there exist σ', t' such that $(\sigma, t) \longrightarrow (\sigma', t')$.*

Theorem 6.2 (Preservation). *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$. Then $(\sigma, t) \longrightarrow (\sigma', t')$ implies that there exists a typing context Δ' such that $\sigma' \sim \Delta, \Delta'$ and $\Delta, \Delta' \vdash t' : T$.*

Proofs of both theorems are attached in an appendix.

6.5 Formalising Capture-Unchecked Terms

I proceed to introduce GradCC, which extends ModCC and allows capture-unchecked code to be formally represented. GradCC borrows inspiration from the gradual typing literature [Siek and Taha 2006; Wadler and Findler 2009; Wadler 2015], where *casts* are used to formally represent type-unchecked code. Concretely, in such a representation the types of all expressions have been erased (i.e., they were all cast to the dynamic type Dyn) and every time an expression is used as an operand, an appropriate cast is used (e.g., an expression is cast to Int before being incremented).

GradCC takes a similar approach: it allows *marking* a path $\# p$, which replaces its capture set with a *mark* $\#$, marking the path as capture-unchecked. Accordingly, capturing types are now equipped with a *capture descriptor* $C?$, which is either a capture set C as in ModCC, or a mark $\#$. Capture sets themselves still only contain proper (non-marked) paths, but they may be *improper* if they can be widened through subcapturing so that they contain a path which is bound at a marked capturing type $S\char`^\#$. By extension, a path p is improper iff the capture set $\{p\}$ is improper.

Since a marked path is capture-unchecked, it allows accessing arbitrary capabilities. Likewise, an improper path allows (indirectly) accessing an actual capture-unchecked, marked path, which similarly may mean accessing arbitrary capabilities.

The following example shows the example capture-unchecked `Logger` package discussed in 6.3.3 may be represented with a GradCC term. Recall that a capture-unchecked package corresponds to a normal Scala package. Importantly, the types in such a package do not mention any capture sets. To integrate this with the capture-aware type system, types occuring in a capture-unchecked are interpreted as marked capturing types S^#, i.e., types whose inhabitants can capture arbitrary capabilities and whose authority should be dynamically enforced. (Types which are well-known to always be pure, such as `String`, do not need to be marked.) Accordingly, capability references in the body of `Logger` correspond to a marked path #fs, a special path form which types any path with a mark # instead of a capture set.

```
#package Logger:                   let logger =
  def log(msg: String): Unit =       let r = region in
    fs.open(...).write(msg)          let _log = λ(msg: String)
                                       let h = #fs.open (...) in
                                       #h.write msg
                                     in mod(#r) { log = #_log }
                                   in ...
```

Dynamic restrictions. A marked path can be used similarly to a capture-checked one, e.g., it can be called if its shape type is a function type. Since its type does not specify a capture set, we no longer know what capabilities may be accessed through the marked path; improper paths pose similar problems. To solve this problem, GradCC features the *enclosed term form* **encl**[C][T] t, which allows dynamically restricting what capabilities may be accessed by t. The capture set C is a *restriction*: it lists the capabilities which may be accessed by t. The restriction C can only contain regions, which correspond to Enclosure memory arenas (Section 6.6.2); this allows enclosed term forms to be efficiently implemented with Enclosures.

Obscuring marks. Capture-unchecked code needs to call out to capture-checked code. For instance, consider the following snippet of Gradient.

```
#package Logger
  def logAll(msgs: List[String])
    msgs.foreach { msg =>
      fs.open(...).write(msg)
    }
```

In this snippet, a capture-unchecked package `Logger` calls `List#map`, a capture-checked function. This is enabled by the *obscur* form **obscur** p **as** x **in** t, which allows *temporarily* treating p as though its capture set was {**cap**}. An obscur form can only be used with a dynamic authority restriction in place. Capture Tracking is used to ensure x is *scoped* and cannot be accessed outside of the dynamic extent of its lexical scope. The `Logger.logAll` definition can be formally represented with the following term.

```
let logger =
  let r = region in
  let _logAll = λ(msg: List[String])
    let f : (String -> Unit)^# = λ(msg: String)
      (#fs).open(...).write(msg)
    obscur f as g in
      msgs.foreach g
  in mod(#r) { log = #_log }
in ...
```

Marks and boxes. Capture-unchecked code also needs to interact with boxes. First, both marked and improper paths are still tracked and need to be boxed before being written to mutable state. Second, as capabilities read out of capture-checked mutable references must be unboxed, improper paths can be boxed *and* boxes can be opened with a mark-open form $\# \multimap p$.

Definition 6.4 (Capture Descriptor Operators)**.** *Capture set operators are extended to capture descriptors as follows. Note that* $\#$ *is effectively empty according to both* $\dot{\in}$ *and* $\dot{\propto}$.

$$
\begin{array}{lll}
\# \mathbin{\dot{\cup}} C? \triangleq \# & \# \mathbin{\dot{\ominus}} x \triangleq \# & p \mathbin{\dot{\in}} C \triangleq p \in C \\
C? \mathbin{\dot{\cup}} \# \triangleq \# & C \mathbin{\dot{\ominus}} x \triangleq \{\, \overline{y.f} \in C \mid y \neq x \,\} & p \mathbin{\dot{\notin}} \# \\
C_1 \mathbin{\dot{\cup}} C_2 \triangleq C_1 \cup C_2 & & x \mathbin{\dot{\not\propto}} \#
\end{array}
$$

Definition 6.5. *The* cv *function is extended as follows. Previous rules use capture descriptor operators instead of capture set operators.*

$$
\begin{array}{lcl}
\mathrm{cv}(\# p) & \triangleq & \# \\
\mathrm{cv}(\textbf{\textit{encl}}[C'][S\,\hat{}\,C?]\; t) & \triangleq & C? \mathbin{\dot{\cup}} C' \\
\mathrm{cv}(\textbf{\textit{obscur}}\, p\; \textbf{\textit{as}}\, x\; \textbf{\textit{in}}\, t) & \triangleq & \# \\
\mathrm{cv}(\# \multimap p) & \triangleq & \#
\end{array}
$$

6.5.1 Changes to the System

Figure 6.7 shows the complete syntax of the new GradCC forms and Figure 6.8 shows the new subtyping and typing rules. The rules make use of the following auxilliary definition.

Definition 6.6 (Well-Formed Restriction)**.** *C is a well-formed restriction in* Γ*, or* $\Gamma \vdash C$ ***wfr****, iff we have* $C = \{\overline{x_i}^{\,i}\}$ *such that* $\overline{\Gamma \vdash x_i : \mathrm{Reg}\,\hat{}\, D_i}^{\,i}$ *for some* $\overline{D_i}^{\,i}$.

The subcapturing rules of GradCC are the same as they were in ModCC, in particular subcap-

Unmarked path	ρ	$::=$	$x.\overline{f}$
Stable path	p, q	$::=$	$\rho \mid \#\rho$
Term	t, u	$::=$	$\ldots \mid \textbf{encl}[C][T]\,t \mid \textbf{obscur}\,p\,\textbf{as}\,x\,\textbf{in}\,t \mid \#\multimapinv r$
Type	T, U	$::=$	$S\,\hat{}\,C?$
Capture descriptor	$C?$	$::=$	$C \mid \#$
Capture set	C	$::=$	$\{\,\overline{\rho}\,\}$

Figure 6.7: GradCC syntax. Unmarked paths ρ are used only to make the syntax more succinct.

turing still only relates capture sets C and not capture descriptors $C?$. Subtyping is extended with (MARKED), which relates marked types. While $S\,\hat{}\,C$ and $S\,\hat{}\,\#$ are unrelated via subtyping, it is always possible to convert a term from $S\,\hat{}\,C$ to $S\,\hat{}\,\#$ by marking it. Typing is extended with four straightforward rules, one per each new term form. Notably, (OBSCUR) only allows returning pure terms from an obscur form, forcing any returned capability to be boxed; doing so ensures that x cannot be accessed during the extent of the obscur form. Additionally, note that the cv of an obscur form is always #, which means that obscur forms can only occur under an enclosure in proper programs.

6.5.2 Reduction

GradCC operational semantics are defined in terms of two reduction relations (Figure 6.9). The "underlying" relation $\cdot \longrightarrow \cdot$ relates two configurations and is an extension of the reduction relation from ModCC. The "primary" relation $\cdot \longrightarrow_e \cdot$ enforces runtime restrictions of enclosures. According to $\cdot \longrightarrow_e \cdot$, a configuration reduces as it normally would (according to $\cdot \longrightarrow \cdot$) iff the redex is permitted in the current restriction; otherwise the configuration reduces to **fail**. The redexes for creating a reference, reading to it or writing to it are permitted only if the involved region is within the current restriction; other redexes are always permitted. These semantics match the behaviour of Enclosures [Ghosn et al. 2021], which stop the program if it tries to access memory outside of the currently imposed restriction.

6.5.3 Metatheory

The statement of soundness for GradCC is a bit more involved compared to ModCC, since well-typed programs may contain capture-unchecked fragments and thus inherently can reduce to **fail**. Such a result signals that a capture-unchecked fragment violated a restriction imposed on it and like in gradual typing systems, such failures should not be prevented. Instead, in addition to the usual properties ensured by the Progress and Preservation theorems, the system should also ensure that capture sets allow predicting what capabilities may be accessed,

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

MARKED
$$\frac{\Gamma \vdash S_1 <: S_2}{\Gamma \vdash S_1{\wedge}\# <: S_2{\wedge}\#}$$

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

ENCLOSURE
$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash C\ \mathbf{wfr}}{\Gamma \vdash \mathbf{encl}[C][T]\ t : T}$$

OBSCUR
$$\frac{\Gamma \vdash p : S{\wedge}C? \qquad \Gamma, x : S{\wedge}\{\mathbf{cap}\} \vdash t : R}{\Gamma \vdash \mathbf{obscur}\ p\ \mathbf{as}\ x\ \mathbf{in}\ t : R}$$

MARK
$$\frac{\Gamma \vdash p : S{\wedge}C?}{\Gamma \vdash \#\, p : S{\wedge}\#}$$

UNBOX-MARK
$$\frac{\Gamma \vdash p : \Box\ S{\wedge}C \qquad \overline{\Gamma \vdash q\ \mathbf{bd}}^{q \in C}}{\Gamma \vdash \# \multimapinv p : S{\wedge}C}$$

Figure 6.8: New subtyping and typing rules of GradCC.

as they did in the original $\mathrm{CC}_{<:\Box}$ system [Boruch-Gruszecki et al. 2023].

The cv function gives us the capabilities referenced by a term. Intuitively, since all access to program and system resources is mediated via capabilities, it should be possible to use cv of a term to predict what capabilities it may access. This also ensures that capture sets assigned in typing are meaningful and indeed allow reasoning about capability access, since typing assigns capture sets based on cv.

GradCC includes regions and mutable state, corresponding to memory accessible by the real Gradient program. Therefore, after widening the cv of a program so that it only contains regions, the resulting capture set gives us an upper bound on the regions accessible by the program. This property is woven into the standard Progress and Preservation Theorems as follows.

Theorem 6.3 (Region-Aware Progress)**.** *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$ such that $\Delta \vdash \mathrm{cv}(t) <: C$ and $\overline{\Delta \vdash r : \mathrm{Reg}{\wedge}\{\mathbf{cap}\}}^{r \in C}$. Then either there exists r such that $t = r$, or $(\sigma; t) \longrightarrow_e$ **fail**, or there exist σ', t' such that $(\sigma; t) \longrightarrow (\sigma'; t')$ and $\mathscr{A}(\sigma, t) \subseteq \{l \mid r \in C, \sigma(r) = \textbf{region}_l\}$.*

Theorem 6.4 (Region-Aware Preservation)**.** *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$ such that $\Delta \vdash \mathrm{cv}(t) <: C$ and $\overline{\Delta \vdash r : \mathrm{Reg}{\wedge}\{\mathbf{cap}\}}^{r \in C}$. Then $(\sigma; t) \longrightarrow (\sigma'; t')$ implies that there exists a typing context Δ' such that $\sigma' \sim \Delta, \Delta'$ and $\Delta, \Delta' \vdash t' : T$ and $\Delta, \Delta' \vdash \mathrm{cv}(t') <: C \cup \{l\}$, where l is the region created during the reduction, if any.*

Reduction $\boxed{(\sigma; t) \longrightarrow (\sigma; t)}$

$$\begin{array}{llll}
(\sigma;\eta[\ C?\ \multimapinv r\ ]) & \longrightarrow & (\sigma;\eta[\ r'\ ]) & \textbf{if}\ \sigma(r) = \Box\, r' \quad (\text{OPEN}) \\
(\sigma;\eta[\ \textbf{encl}[C][T]\, r\ ]) & \longrightarrow & (\sigma;\eta[\ r\ ]) & \quad (\text{EXIT}) \\
(\sigma;\eta[\ \textbf{obscur}\ r\ \textbf{as}\ x\ \textbf{in}\ t\ ]) & \longrightarrow & (\sigma;\eta[\ [x := r]\, t\ ]) & \quad (\text{OBS})
\end{array}$$

$$\textbf{Eval context}\quad \eta \quad ::= \quad \ldots \quad | \quad \textbf{encl}[C][T]\,\eta$$

$\boxed{(\sigma; t) \longrightarrow_e ((\sigma; t) \mid \textbf{fail})}$

$$\frac{(\sigma;\eta[\ t\ ]) \longrightarrow (\sigma';\eta'[\ t'\ ]) \qquad \mathscr{A}(\sigma, t) \subseteq \mathscr{R}(\sigma,\eta)}{(\sigma;\eta[\ t\ ]) \longrightarrow_e (\sigma';\eta'[\ t'\ ])} \qquad \frac{\mathscr{A}(\sigma, t) \not\subseteq \mathscr{R}(\sigma,\eta)}{(\sigma;\eta[\ t\ ]) \longrightarrow_e \textbf{fail}}$$

$$\mathscr{R}(\sigma, \textbf{encl}[\{\overline{r_i}^{\,i}\}][T]\,\eta) \triangleq \{\overline{l_i}^{\,i}\} \cap \mathscr{R}(\sigma,\eta) \qquad \textbf{if}\ \overline{\sigma(r_i) = \textbf{region}_{l_i}}^{\,i}$$

$$\mathscr{R}(\sigma, \textbf{let}\ x = \eta\ \textbf{in}\ t) \triangleq \mathscr{R}(\sigma,\eta) \qquad \mathscr{R}(\sigma, [\,]) \triangleq \{\}$$

$$\begin{array}{llll}
\mathscr{A}(\sigma,\eta[\ r.\textbf{ref}\ r'\ ]) & \triangleq & \{l\} & \textbf{if}\ \sigma(r) = \textbf{region}_l \\
\mathscr{A}(\sigma,\eta[\ !r\ ]) & \triangleq & \{l\} & \textbf{if}\ \sigma(r) = l \triangleright \textbf{ref}\ v \\
\mathscr{A}(\sigma,\eta[\ r := r'\ ]) & \triangleq & \{l\} & \textbf{if}\ \sigma(r) = l \triangleright \textbf{ref}\ v \\
\mathscr{A}(\sigma,\eta[\ t\ ]) & \triangleq & \{\} & \textbf{if}\ t\ \text{is a different redex form}
\end{array}$$

Figure 6.9: GradCC operational semantics.

The above theorems form the intended statement of correctness for GradCC. I attach proofs of the following theorems in an appendix.

Theorem 6.5 (Preservation)**.** *Let* $\sigma \sim \Delta$ *and* $\Delta \vdash t : T$*. Then* $(\sigma; t) \longrightarrow (\sigma'; t')$ *implies that there exists a typing context* Δ' *such that* $\sigma' \sim \Delta, \Delta'$ *and* $\Delta, \Delta' \vdash t' : T$*.*

Theorem 6.6 (Progress)**.** *Let* $\sigma \sim \Delta$ *and* $\Delta \vdash t : T$*. Then either there exists* r *such that* $t = r$*, or there exist* σ', t' *such that* $(\sigma; t) \longrightarrow (\sigma'; t')$*.*

The proofs are carried out for a version of GradCC without the obscur form **obscur** p **as** x **in** t; I only provide an intuitive argument for the soundness of the complete system. First, an obscur form can only be used within the dynamic extent of an **encl** form, since the cv of an **obscur** form is always #. Hence, it is sound to access p immediately within the obscur form even if it is improper. Second, any capability returned from an obscur form must be boxed (see (OBSCUR)), which thanks to Capture Tracking ensures x cannot be accessed outside of its lexical scope [Boruch-Gruszecki et al. 2023]. I expect the complete proof to be straightforward.

6.6 Evaluation

6.6.1 Migrating the Scala XML Library

I migrated `scala-xml`, the standard Scala XML library [ScalaXML 2023], to a capture-checked Gradient package. The `scala-xml` library was chosen since most of its code does not need to access any system resources, with the primary exceptions being the XML parser (which may need to resolve DTDs from the filesystem or from the network), and the convenience functions for, e.g., loading and parsing an XML file. Still, occurences of capabilities in the codebase were more common than expected:

- code for rendering an XML object into a `String` was implemented by manipulating mutable `StringBuffer`-s,
- some classes representing XML data had mutable fields, contrary to what idiomatic Scala code would do, and
- some functionality was implemented by calling Java code, e.g., parsing XML.

Migrating the library also revealed the need for the formalism to distinguish between records and modules and to separate regions from modules. The former allows understanding class instances as potentially pure records, as opposed to always-impure (tracked) modules; the latter supports local mutable state.

Despite these difficulties, migrating `scala-xml` to a capture-checked Gradient package required few changes to the codebase. The library has 4200 LoC (excluding comments); adding capture annotations to it required modifying c. 260 LoC and involved no refactoring. Most of the changed lines (c. 200) are similar to the following example, i.e., the change involves merely adding a few extra characters.

```
// before the migration
def buildString(sb: StringBuilder): StringBuilder
// after the migration
def buildString(sb: StringBuilder^): StringBuilder^{sb}
```

Our `scala-xml` experiment shows that migrating a real-world Scala codebase to a non-ocap, capture-checked Gradient package is not a significant amount of effort; such migrations are a valuable intermediate step on the way to migrating a codebase to an ocap module. A full migration may require significantly refactoring the codebase so that it receives all the devices it needs as arguments from its callsites, and will likely require the users of the codebase to adjust their code too. At the same time, ocap modules are more flexible, since they allow their users to *attenuate* the authority of capabilities passed to the module (Section 6.3.1).

I attach the migrated sources and a migration report as supplementary material. The report lists the steps I took to migrate the library, explains how I understand Scala features such as classes and packages in terms of the formalism I presented, and suggests how to verify I migrated the library correctly.

6.6.2 Implementing Gradual Compartmentalization

I outline the major steps to extending an existing language with gradual compartmentalization.

Add object capabilities and modules as an extension

The Gradient support for these features was outlined in Section 6.3. To ensure capability safety, it may be necessary to additionally restrict or *tame* [Miller 2006] existing language features, e.g., an implementation of Gradient would need to tame the Scala standard library (potentially by assigning it appropriate capture signatures) and restrict ocap code from using features such as Java reflection. The necessary work for the Java case was studied by the authors of Joe-E [Mettler et al. 2010] and Wyvern [Melicher 2020]. There are many other examples of ocap extensions for existing languages in the literature, e.g., the Caja extension for Javascript [Miller et al. 2008], the Emily extension for OCaml [Stiegler 2007; Stiegler and Miller 2006], the CaPerl extension for Perl [Laurie 2007], and the Oz-E extension for Oz [Spiessens and Van Roy 2005].

Track capabilities in the type system

Gradient uses Capture Tracking to track the authority of objects in their types, as was formalised in Section 6.4 and Section 6.5. The most essential reason for tracking capabilities in types is letting ocap code interact with non-ocap code by using the type system to ensure system access restrictions can be enforced (Section 6.3).

The non-essential (although still important!) reasons for using Capture Tracking in Gradient include the borrow safety property (6.3.2), the minimal notational burden imposed by the approach (Section 6.6.1), and a preexisting Capture Tracking implementation in the Scala compiler.

Even though Gradient relies on Capture Tracking, another implementation of gradual compartmentalization could use a different approach for tracking the authority of objects in their types. Such approaches already exist in the literature: Lee et al. [2023] show an alternative version of Capture Tracking without dependent types, and Brachthäuser et al. [2022] show a system for tracking capabilities in types without subtyping. Naturally, they come with tradeoffs: signatures of capture-polymorphic definitions become more unwieldy without dependent types, and giving up subtyping means giving up subcapturing and its connection to the capability derivation hierarchy.

Add support for dynamic capability access restrictions

Gradient's `enclosed` block can be implemented via the LitterBox framework built for Enclosures [Ghosn et al. 2021], which relies on hardware support.[4] Gradient regions and devices (Section 6.3.3) *directly* correspond to Enclosure memory arenas and system call restrictions. Relying on hardware support allows Gradient code to impose access restrictions even across FFI calls to binary code which can forge pointers. Furthermore, compartmentalizing a real-world application such as a web server via Enclosures has an acceptable performance cost; depending on the hardware used, the slowdown factor can be as small as 1.02 [Ghosn et al. 2021].

The access checks of `enclosed` blocks could be carried out in software, if we disallowed FFI and accepted worse performance. A special compilation scheme would be necessary, where all access to system features and mutable state goes through methods which inspect if the access is currently permissible; checks on access to other objects are not necessary. This scheme assumes that the only way to invoke system calls from within the language is by accessing devices (i.e., objects whose methods are language primitives which actually invoke system calls).

Conclusions

Extending an existing language with gradual compartmentalization is an effort of a similar magnitude to implementing a new ocap language. Gradual compartmentalization rests on solid foundations [Mettler et al. 2010; Melicher 2020; Boruch-Gruszecki et al. 2023; Ghosn et al. 2021]; the tasks necessary for implementing it were studied independently for different contexts and are well understood.

[4]Concretely, two backends for Enclosures were implemented, one using Intel VT-x (a widespread virtualization feature) and another using Intel MPK.

7 Background

In this section I describe the existing literature related to Capture Tracking. Given the vast amount of existing work, the description is necessarily incomplete; if there is something missing from this section, it is through my own fault.

First, I provide a sweeping overview of the background on which Capture Tracking builds. The point of the overview is to highlight particular systems and approaches and list their most salient aspects, i.e., to present, as much as is reasonable, the design space explored in existing literature. Next, I compare Capture Tracking with its closely related works, and afterwards I do the same for Gradual Compartmentalization.

7.1 Background

7.1.1 Effects

Effect systems assign *effects* to terms, in addition to types. The typing judgment in such a system typically is of the form $\Gamma \vdash t : T \mid \chi$, where χ describes the overall effects of evaluating t. Types of abstractions in such systems need to be augmented with a *latent* effect, e.g., $S \rightarrow^{\chi} T$, where χ describes the deferred effects of the abstraction's body. Effects can be seen as dual to capabilities: the former allow terms to contain arbitrary operations and assign an effect to each term, while the latter only allow terms to invoke effectful operations by accessing capabilities bound in the context; capabilities are thus closer to coeffects [see Petricek et al. 2014a] than to effects.

The earliest effect systems I am aware of track reading from, writing to and allocating mutable references [Gifford and Lucassen 1986; Lucassen and Gifford 1988]. The later system extends the earlier one with with regions and effect polymorphism. An effect system may include a notion of *subeffecting*, first proposed by Talpin and Jouvelot [1992]. Region-based memory management systems proposed by Tofte and Talpin [1997] (MLKit) and Grossman et al. [2002] (Cyclone) employ an effect system to ensure that region-associated references captured by closures are not accessed after the region is deallocated. Some other applications of effect systems include ensuring safe error handling (“checked exceptions”) [Gosling et al. 2014], various safety properties in a concurrent setting [Boyapati and Rinard 2001; Gordon 2017], strong atomicity in a transactional memory setting [Abadi et al. 2008], and purity [Fähndrich et al. 2006; Hunt and Larus 2007; Pearce 2011]. Wadler and Thiemann [2003] show that there is a tight connection between effect systems and monads [Moggi 1991]; monads as implemented in Haskell [Peyton Jones and Wadler 1993] are, for most intents and purposes, an effect system.

Effect handlers [Plotkin and Pretnar 2013] are closely connected to algebraic effects, introduced by Plotkin and Power [2003]. Algebraic effects can be understood as a restriction on general monads which, unlike monads, can be freely composed. Effect handlers themselves can be understood as resumable exceptions, or otherwise as a structured way of exposing delimited continuations. Koka [Koka 2023; Leijen 2014, 2016] is a language with support for algebraic effects and effect handlers. It features an effect system based on row polymorphism with potentially duplicated effects; the system allows *masking* effects which are local to a particular subterm, such as internally throwing and handling an exception. Koka is also an example of a system which tracks *divergence* as an effect. The Effekt language [Brachthäuser et al. 2020b,a, 2022] features *capability-based* effect handlers, similar to the ones presented in this thesis (Section 3.3.5).

Effect quantales [Gordon 2017, 2021] are a generic, effect-polymorphic system of *sequential* effects. A sequential effect system [Tate 2013] is an effect system where only particular effect sequences are allowed to occur; such systems thus feature a degree of flow-sensitivity. Such a system can be used, for instance, to ensure that synchronization locks are used correctly. Notably, *store-sensitive effect quantales* were proposed and used by Bao et al. [2021] to track mutation and model move semantics. A number of other generic effect systems were proposed [Marino and Millstein 2009a; Tate 2013; Rytz et al. 2012]

7.1.2 Resource Ownership

Resources are entities whose identity or provenance (i.e., approximate identity) matters. We may care about *what* resource may or may not be accessed, or we may care about a resource being allocated and de-allocated. In most systems, this is tracked with an *ownership* system, which restricts how resources may be aliased.

Linear type systems are perhaps the most basic example of a resource ownership system. Briefly, a linear variable can be used precisely once. They were introduced by Wadler [1990] as a way to extend a functional programming language with support for local mutable data without violating referential transparency. Linearity is closely related to uniqueness; Marshall et al. [2022] present a type system featuring both concepts and gives a detailed account of their similarities and differences.

Ownership types, as originally proposed, allow objects to encapsulate particular references.

For instance, a linked list can encapsulate references to its nodes, ensuring details about its internal representation are not leaked. They were originally proposed by Clarke et al. [1998] and Clarke [2002], based on “Flexible Alias Protection” of Noble et al. [1998]. The property guaranteed by the original ownership types has been explained as “owners-as-dominators”, i.e., ensuring that references to the owner should dominate references encapsulated by the owner in the reachability graph of objects. The literature features a vast amount of works presenting systems derived from ownership types and using them to solve strikingly diverse problems such as concurrency control, memory management and conformance to software architecture specifications [Aldrich et al. 2002]. In particular, they were also used to control effects [Clarke and Drossopoulou 2002]. Clarke et al. [2013b], Clarke et al. [2013a], and Mycroft and Voigt [2013] survey the literature on ownership types.

Separate uniqueness was proposed by Haller and Odersky [2010] as heap structuring invariant where “unique” references form subgraphs disjoint from the rest of the object graph. In contrast to earlier works on *external uniqueness* by Clarke and Wrigstad [2003], separate uniqueness ensures sending unique references across thread boundaries does not need any additional restrictions imposed on the references.

Rust [Rust 2023] features an ownership discipline which is arguably the most closely related to linear types. Rust features a memory management scheme where all allocations are owned by a single entity, either a binding on the stack or another allocation. At any point, there is only a single unique reference “owning” a particular allocation, thus effectively partitioning the heap into a tree. Stacked Borrows [Jung et al. 2019] are a system of operational semantics which accurately formally models memory access guarantees provided by Rust. Jung et al. [2017] show a more classical formal calculus for Rust, with a particular focus on proving safety in presence of code using Rust’s `unsafe` blocks, which are particularly challenging as they temporarily disable safety checks to allow more flexible code than is otherwise possible. In contrast, Pearce [2021] shows a more lightweight formal calculus which still captures most salient aspects of Rust, including its ownership discipline; the publication features a remarkably detailed description of related work.

Reggio [Arvidsson et al. 2023a,b] is the region-based memory management scheme of Verona. Reggio regions can be freely created. A single external pointer to a *bridge object* allocated on the region is allowed, while a region’s interior can have arbitrary aliasing; effectively, such regions form a tree. The bridge object can be swapped for a different same-region object. Regions can be visited via their bridge objects using a push/pop *region stack*: only the topmost region can be mutated, other regions on the stack are read-only (which excludes allocations), regions outside the stack are completely inaccessible. Additionally, a region can only be visited by a single thread. This scheme enables an absence of data races and cheap exchange of region ownership. Additionally, each region can have its own memory management scheme, facilitating a flexible although experimental approach to memory management. The most salient distinguishing feature of Reggio (perhaps also the most experimental) is that at any point, only a *single* Reggio region can be mutated.

Borrowing, broadly speaking, temporarily relaxes the aliasing restrictions otherwise present in the system. Rust's borrows are perhaps the most broadly known occurence of the concept. In Rust, borrowing a reference temporarily disables it and creates a new reference pointing to the same entity; the original reference is *disabled* until the borrow goes out of scope. The borrow can be either mutable or shared (immutable); shared borrows can be duplicated as long as they all go out of scope simultaneously. At any point, there only exists either a single *active*[1] mutable reference, or multiple *active* immutable ("shared") references. Finally, a borrowed reference can only be stored in allocations which it outlives. This scheme, reminiscent of fractional permissions [Boyland 2013], enforces a multiple-reader/single-writer model of concurrency. Jung et al. [2019] presents operational semantics which account for Rust's borrowing/aliasing model.

The linear type system proposed by Wadler [1990] features an early form of borrowing, where a linear variable standing for mutable data can be temporarily treated as read-only, allowing it to occur multiple times in a particular subterm. Other examples of systems featuring borrowing include the works of Boyland and Retert [2005] and Radanne et al. [2020].

Framing is a feature of separation logic [Reynolds 2002; O'Hearn 2019] which is rather similar to borrowing. Separation logic, a descendant of Floyd-Hoare logic, allows reasoning about shared mutable data structures, and is built around reasoning about separated (i.e., disjoint) parts of the heap. To facilitate such reasoning, separation logic allows temporarily *framing away* certain unused references, so that local references can be seen as pointing to a separated heap section.

7.1.3 Capabilities

The capability approach allows controlling access to sensitive functionality. In this model, sensitive operations can only be performed by possessing and invoking an appropriate capability. A good example of a capability-like entity is a Unix file handle: the primary means through which a Unix program accesses the disk is by creating and using an appropriate file handle. Furthermore, the capability model forbids *ambient authority*: an agent can only gain access to an existing capability if it receives it from another agent. If Unix was designed following the capability model, then the only way for a program to create a file handle would be by using another capability, e.g., by listing the contents of a directory the program already has a handle for. In addition, programs would need to start with no capabilities by default, and would need to receive the capabilities they need from their outside context.

The topic of this dissertation revolves around the *object capability* model, where capabilities are special objects which allow accessing sensitive functionality simply by calling their methods, and having a capability object is the same as being able to use it.

There is a long line of research into the object capability model. Dennis and Van Horn [1966]

[1]Technically, borrows can be *re-borrowed*: the reference from which a borrow is created can itself be a borrow.

discuss how *capabilities* are used in the security model of a *multiprogrammed computer system*, which we would now call a time-sharing system. In particular, the *secure entry points* discussed by Dennis and Van Horn bear many similarities to object capabilities, although they are not objects per se. Morris [1973] explicitly points out that the usual treatment of closures (specifically, procedures paired together with an environment which cannot be introspected), as black boxes has multiple benefits for ensuring a program's correct operation even if one of its components is faulty, and W7 [Rees 1996] is an early example of a programming language with built-in support for capabilities.

The seminal thesis of Miller [2006] on the E language was the first to propose "object capabilities" as the name for the technique of identifying capabilities with objects; it provides a detailed description of the benefits of the capability model and the advantages of unifying capabilities and objects. The *idea* of object capabilities predates the work of Miller: anecdotal information suggests it was previously known in the industry, and some of its most direct previous occurences in the literature are in the works of Levy [1984] (see Section 10.5, where identifying objects with capabilities is considered) and Chase et al. [1992] (see the introduction: "a client must possess an unforgeable object reference for the service, and can only operate on the service by invoking its methods").

Capabilities facilitate enforcing the *Principle of Least Authority*. PoLA is analogous to the Principle of Least Privilege [Saltzer 1974]. However, the latter is based on an intuitive but imprecise notion of "Privilege", whereas, following Miller [2006], "Authority" is defined as all the effectful operations which may be invoked by a particular object. More broadly, I will also speak of the authority of an agent or a principal or a process, with the same meaning in mind; in the object capability model, it is most convenient to unify all of these concepts with objects, but in practice it does not truly matter if they are technically a distinct sort of entity or not.

PoLA states that any entity's authority should be restricted to the minimum necessary to carry out its functionality. The Unix `cat` program violates PoLA: it executes with all the privileges of the user invoking the program, even though it only needs to read the contents of a few files and write to the console. In a capability-safe setting, `cat` would receive the capabilities it needs when it is invoked.

The Confused Deputy Problem [Hardy 1988] is another of the original motivating examples for capabilities [Rajani et al. 2016]; I find the example particularly illustrative and so, I describe it in detail. The example is based on a real-life situation encountered by Tymshare, a company providing "commercial timesharing" services, whose operating system had similar protection structures to Unix. The system included a Compiler, in a particular System Directory. The Compiler was writing telemetry (information about what compiler features were used) to a log stored in the System Directory. Since the directory was protected and users could not normally write to it, the compiler file was marked as having a special "home files license" which let it write to arbitrary files in the System Directory.

Some users found an "interesting" loophole in the design: they discovered that they could instruct the Compiler to overwrite files in the System Directory by telling it to write its debugging output to those files. They came to know that the System Directory stored a file with billing information, and they used the loophole to overwrite the file, causing their billing information to become lost in the process.

What went wrong? The Compiler was the eponymous "Confused Deputy": it was serving two masters, the Company and the User. The Company allowed the compiler to write to a sensitive system location, while the User let the compiler write to their own personal files. This was expressed as having the Compiler execute with a permission to write to *both* sorts of locations indiscriminately, without regard for whose orders the Compiler was acting on.

What is notable about this example is that PoLA alone would not truly prevent it: the Compiler really had a need to write its telemetry to a file in the System Directory, and the User should not be able to instruct the Compiler to overwrite this file. Still, the capability model *does* offer a natural solution: the Compiler should start with a capability to write its telemetry, and should receive capabilities to write its (real and debug) output from the User. First, since writing to a file requires invoking a particular capability, the Compiler needs to explicitly state on whose authority it is writing its debug output, naturally distinguishing between the two "masters". Second, since in a capability setting an agent cannot create a capability they are not allowed to use, the User would be prevented from creating the capability to write to a file in the System Directory in the first place.

Numerous *ocap dialects* were developed for pre-existing languages following the path blazed by Miller [2006], e.g., the Caja extension for Javascript [Miller et al. 2008], the Emily extension for OCaml [Stiegler 2007; Stiegler and Miller 2006], the CaPerl extension for Perl [Laurie 2007], and the Oz-E extension for Oz [Spiessens and Van Roy 2005]. Such dialects focused on *taming* [Miller 2006] existing language features to provide an ocap-safe extension, as opposed to integrating ocap and non-ocap code in a single codebase like gradual compartmentalization does.

The Wyvern language features object capabilites, including a module model [Melicher et al. 2017] which was used as a building block for Gradient, as well as an effect system for controlling access to capabilities [Melicher 2020]. Melicher et al. [2017] present a safety theorem for reasoning about authority of objects by reasoning about the *permission* (using terminology of Miller [2006]) to directly access capabilities; using a more sophisticated framework, direct, rigorously formal reasoning about authority and attenuation is possible [Devriese et al. 2016].

Reference capabilities are a scheme for restricting the authority of object capabilites via their types. The term was first proposed by Clebsch et al. [2015], although Boyland et al. [2001] show a similar technique applied in a different context. I present the idea in my own words, using terminology derived from [Miller 2006]. Reference capabilities quantify capability types with a *permission*. Such permissions, much like the ones we know from Unix, describe what operations can be invoked using the capability. For instance, a permission may only allow read-only access to a mutable object, or it may only allow reads and writes, but not deallocation.

They were employed in Reggio [Arvidsson et al. 2023a,b] to restrict which objects can be currently mutated. Gordon et al. present a system of reference capabilities which ensures an absence of data races and deterministic execution while permitting threads to exchange references to mutable data [Gordon et al. 2012b,a]; the system was used as the formal basis for a prototype extension to C$^\sharp$ validated in practice by a large team at Microsoft.[2]

7.2 Related Work: Capturing Types

Effects as Capabilities. Establishing effect safety by moderating access to effects via term-level capabilities is not a new idea [Marino and Millstein 2009b]. 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, Capture Tracking uses 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 [Zhang and Myers 2019; Biernacki et al. 2020; Brachthäuser et al. 2020b] but those can result in overly verbose and difficult to understand types [Brachthäuser et al. 2020a]. A third approach, taken by Capture Tracking, 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

[2]This was the team working on the Midori project, as confirmed in personal communication with Colin S. Gordon. See also https://joeduffyblog.com/2015/11/03/blogging-about-midori/.

for tracking captured capabilities, which can in specific cases be more precise than cv. In comparison, $\mathsf{CC}_{<:\Box}$ 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 $\mathsf{CC}_{<:\Box}$. Whereas in $\mathsf{CC}_{<:\Box}$ 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 $\mathsf{CC}_{<:\Box}$ 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 and separation can be tracked in $\mathsf{CC}_{<:\Box}$.

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, Capture Tracking supports 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 $\mathsf{CC}_{<:\Box}$, 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 $\mathsf{CC}_{<:\Box}$ – 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 $\mathsf{CC}_{<:\Box}$ 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 $\mathsf{CC}_{<:\Box}$bindings with $\{\mathbf{cap}\}$ 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 $x :: A[¶si] \in \Delta$. Just like our definition of captured variables in $\mathsf{CC}_{<:\Box}$, 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, $\mathsf{CC}_{<:\Box}$ 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 $\mathsf{CC}_{<:\Box}$ 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, Capture Tracking uses 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 $\Gamma @ \mathscr{C} \vdash e : \tau$. Such systems can be seen as similar in spirit to $\mathsf{CC}_{<:\square}$, where additional information is available about each variable in the environment through the capture set of its type. Petricek et al. [2014b] 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, $\mathsf{CC}_{<:\square}$ is based on $\mathsf{System\ F}_{<:}$ (thus supporting type polymorphism and subtyping) and supports capture-polymorphic functions.

Object Capabilities.

The (object-)capability model of programming [Crary et al. 1999; Boyland et al. 2001; 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 Newspeak language [Bracha et al. 2010] features object capabilites. In particular, it features the *platform capability*, an object which grants access to the underlying platform and allows resolving modules and capabilities. The platform capability is similar to the root capability **cap**: a $\mathsf{CC}_{<:\square}$ value assigned the capture of {**cap**} has the authority to access arbitrary capabilities, while capturing the Newspeak platform capability grants access to the entire platform.

The Wyvern language [Melicher et al. 2017] implements the object capability 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 which tracks access to capabilities in the type of methods.

Figueroa et al. [2016] show an intricately engineered encoding of object capabilities in Haskell, where a Haskell module needs to possess appropriate capabilities in order to call a monad transformer's private operations. The capabilities may be organized into a hierarchy, e.g., a ReadWrite capability may subsume the Read and Write capabilities. Capabilities may be shared between modules through encoded friend declarations; a module's authority may be determined like in Wyvern.

In $\mathsf{CC}_{<:\square}$, one can statically reason about authority of capabilities simply by inspecting what capture sets capabilities are typed with. Additionally, subcapturing naturally allows defining capability hierarchies. If we model modules with abstractions, the abstraction's capture set directly reflects its authority. Importantly, $\mathsf{CC}_{<:\square}$ tracks mention rather than use and does not include a separate effect system.

7.3 Related Work: Gradual Compartmentalization

Compartmentalization solutions exist on a spectrum. They range from static support for explicit security policies directly within a language's semantics and types, to dynamic ones enforced by the operating system on arbitrary code. Each design point presents different trade-offs between (1) the expressiveness and granularity of user-defined security policies and (2) the burden put on the programmer to correctly compartmentalize untrusted code. All solutions aim at limiting untrusted software component's access to the rest of the application and system resources.

Neither is Capture Tracking a *sui generis* concept. Many of the underlying ideas were and are studied in the literature. This section first explores different compartmentalization approaches and highlights their trade-offs, and next discusses the literature on tracking capabilities in types.

7.3.1 Static Compartmentalization

Object capabilities. There is a long history of research on object capabilities. As early as 1973, Morris described various language features which can support local reasoning about security properties. W7 is an early example of a language with support for capabilities [Rees 1996]. The seminal thesis on the E language [Miller 2006] may have been the first to explicitly recognize and define the object capability approach, as well as provide a detailed description of its benefits. E inspired many other works on restricting existing languages to build a capability-safe subset [Mettler et al. 2010; Miller et al. 2008; Stiegler and Miller 2006; Laurie 2007; Spiessens and Van Roy 2005].

Gradient's approach to modules is very closely inspired by Wyvern [Melicher et al. 2017], which itself is inspired by Newspeak modules [Bracha et al. 2010] and their predecessors, such as MzScheme's Units [Flatt and Felleisen 1998].

Object capabilities together with a module system enable an application to compartmentalize its components and control their access to program and system resources in an intuitive and familiar way. However, they assume that the application's code is uniformly written assuming no ambient authority, which is not true of the vast majority of currently existing code.

Programming Languages. Rust allows circumventing its memory safety guarantees within `unsafe` blocks. The motivation for this feature is that circumventing the guarantees is occasionally necessary for expresiveness and that the blocks themselves can easily be located by tooling. In practice, developers make mistakes: Bae et al. [2021] built a tool for automatically scanning the Rust ecosystem for vulnerabilities and identified 264 previously unknown memory safety bugs (leading to 76 CVEs). Moreover, combining safe and unsafe languages in a single application can lead to Cross-Language Attacks [Mergendahl et al. 2022], which would have been prevent by the checks of either language alone, static or dynamic. Preventing such vulnerabilities is one reason to only allow executing unsafe code if its behaviour can be dynamically controlled and restricted, as gradual compartmentalization and Gradient propose.

PCC & Language Virtual Machines. Proof-carrying code (PCC) [Necula 1997; Appel 2001] is an approach which attaches a formal proof to a software component. The proof is checked at load-time to ensure the component adheres to the desired security policies. Certain compartmentalization solutions, such as domain specific languages (e.g., eBPF [McCanne and Jacobson 1993]), compiler instrumentation (e.g., NaCl [Yee et al. 2009]), or even language virtual machines (e.g., WASM [Haas et al. 2017]) can be see as variations of PCC. While such mechanisms work on the level of bytecode, employing them may still require refactoring code, e.g., eBPF code is required to terminate [McCanne and Jacobson 1993]. They further often target specific ecosystems (e.g., web browsers or kernel module subsystems) and require non-negligible efforts to be adapted to other environments [WASI 2023; WASM-Web 2023; WASM-JS 2023].

Libraries. RLBox is a library which aids with compartmentalizing a library via NaCl or WebAssembly, used to compartmentalize the Firefox rendering process at a very fine granularity [Narayan et al. 2020]. While the library is a very useful fit for isolating libraries which need little access to program or system resources (such as image manipulation libraries used in web browsers), the approach shares the issues of NaCl and WebAssembly: Interfacing with program and system resources is difficult and not verified statically.

7.3.2 Dynamic Compartmentalization

Processes. Processes are the default mechanism to isolate applications in a time-sharing operating system. They have been used to compartmentalize applications such as web browsers [Chromium 2023; Mozilla 2023]. They are a clear boundary around untrusted code that encompasses all of the code's resources and has a clear interface to the underlying system to interpose on system calls.Process-based compartments further have the benefit of supporting arbitrary, pre-compiled binaries.

Most applications, however, assume a shared heap and stack and the ability to directly call their libraries. Compartmentalizing existing applications with processes thus requires heavy refactoring so that untrusted libraries are only directly accessed within a separate process. It incurs non-negligible overheads to turn direct calls into synchronous inter-process communication, requires marshalling arguments between processes, and generally increases resource consumption, either through system metadata or duplication of common code dependencies.

OS abstractions. Several solutions [Bittau et al. 2008; Litton et al. 2016; Hsu et al. 2016] extend operating systems with intra-address-space isolation mechanisms. Light-weight Contexts (lwC) [Litton et al. 2016] let application create intra-process compartments with limited access to the program's resources. Despite being more flexible than processes, such solutions still require modifying applications. As these are generally implemented at the system-level, they do not leverage program-specific semantic knowledge and push the burden of compartmentalization onto the programmer. The lack of a clear migration path to compartmentalized applications may in part explain why none of these solutions made its way into mainstream operating systems.

Hardware Extensions. Application compartmentalization operates at a different spatial and temporal granularity than processes. As a result, hardware security extensions appeared to provide hardware-enforced isolation at either (1) finer-granularity (e.g., Mondrian memory at byte-level [Witchel et al. 2002]), (2) with lower temporal overheads (e.g., Intel Memory Protection [Intel 2020] or VmFunc in Intel VT-x [Uhlig et al. 2005]) to switch between compartments, or (3) both (e.g., CHERI [Woodruff et al. 2014]).

Similarly to OS mechanisms, these solutions require either heavily modifying existing applications, or implementing new software development toolchains [Hedayati et al. 2019; Vahldiek-Oberwagner et al. 2019; Lind et al. 2017; Ghosn et al. 2021] (i.e., compilers, standard libraries, runtime environments). The second approach allows leveraging language or application-specific knowledge to (partially) automate code compartmentalization, reducing the migration burden. For example, Enclosures [Ghosn et al. 2021] expose a flexible programming abstraction. They rely on the compiler and the runtime to bridge the gap between programming language constructs and hardware entities. The language runtime transparently create and orchestrates compartment transitions. Enforcing isolation via hardware mechanisms allows Enclosures to support heterogeneous environments. Despite their acceptable performance overheads, Enclosures only detect policy violations at run-time. This can sometimes lead to a costly trial and error to tune restrictions applied to a particular closure and slows down the development process.

7.3.3 Tracking Capabilities in Types

Access to capabilities can be tracked with an effect system. For instance, in the region-based memory management system proposed by Tofte and Talpin [1997], an effect system tracks access to regions. Indeed, practically *any* effect system, starting from the seminal work of Lucassen and Gifford [1988], can be used to track capability access. Such systems were also integrated with object capabilities, e.g., Wyvern features an effect system which allows tracking capability access at method granularity.

However, it has been observed multiple times (e.g., by Osvald et al. [2016] and Brachthäuser et al. [2022]), that the form of polymorphism present in most such systems leads to verbose type signatures, which are arguably the key factor which impeded their broader adoption [Boruch-Gruszecki et al. 2023]. Capture Tracking instead fully relies on intuitive capability-based reasoning: all capabilities within current scope can always be accessed, without needing to state so within the type system. The type system instead tracks if capabilities are returned from scopes that received them, which arguably is also more intuitive: instead of needing to ask for permission when invoking effectful operations as in effect system, types in Capture Tracking clarify which objects *may* be used to access tracked resources and therefore merit particular attention. In practice, Capture Tracking can be retroactively applied to a codebase with a relatively small burden (Section 6.6).

Systems which track capture of particular tracked objects were proposed: the type-based escape analysis of Hannan [1998] and the Open Closure Types of Scherer and Hoffmann [2013]. Neither system features a lightweight polymorphism mechanism similar to Capture Tracking. Rytz et al. [2012] present a type-and-effect system which allows typing higher-order functions with simple signatures without effect polymorphism. Compared to Capture Tracking, the expressivity of the system is limited: it is impossible to type a function whose result's effect is relative to the function's argument, e.g., the function composition operator. *Coeffect systems* augment bindings in the typing context with additional information, not unlike how Capture Tracking augments the type of each bindings with a capture set. Petricek et al. [2014a] show a general coeffect framework which can be used to track various functionality, including data access. In contrast to Capture Tracking, this framework is based on simply typed lambda calculus and does not support coeffect-polymorphic function types.

A $\mathrm{CC}_{<:\square}$ Proofs

A.1 Proof devices

We extend type well-formedness to environments:

Well-formed environment $\boxed{\vdash \Gamma\ \textbf{wf}}$

$$\frac{\vdash \Gamma\ \textbf{wf} \quad \Gamma \vdash T\ \textbf{wf}}{\vdash \Gamma, x : T\ \textbf{wf}} \qquad \frac{\vdash \Gamma\ \textbf{wf} \quad \Gamma \vdash T\ \textbf{wf}}{\vdash \Gamma, X <: T\ \textbf{wf}} \qquad \vdash \varnothing\ \textbf{wf}$$

To prove Preservation (**??**), 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 $\boxed{\Gamma \vdash \sigma \sim \Delta}$

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

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 $\Gamma \vdash \eta[\ t\ ] : T$.*

Fact 1. *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 2. *The analogous holds for $\eta[\ t\ ]$.*

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 $\mathrm{cv}(R \mathbin{\hat{}} C) = C$.

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

Lemma A.1 (Evaluation context typing inversion). *$\Gamma \vdash \eta[\, s\,] : T$ 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 $e = \textbf{let}\, x = e' \,\textbf{in}\, t$. Proceed by induction on the typing derivation of $\eta[\, s\,]$. We can only assume that $\Gamma \vdash \eta[\, s\,] : T'$ 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 $\Gamma \vdash \eta[\, s\,] : T''$ and $\Gamma \vdash T'' <: T'$. We can conclude by the inner IH and (TRANS).

□

Lemma A.2 (Evaluation context reification). *If both $\Gamma \vdash e : U \Rightarrow T$ and $\Gamma \vdash s : U$, then $\Gamma \vdash \eta[\, s\,] : T$.*

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

Lemma A.3 (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* $\sigma = \sigma'[\, \textbf{\textit{let}}\, x = v\, \textbf{\textit{in}}[\,]\,]$. Then $\Delta = \Delta', x : U$ for some U. Since $x \notin \mathrm{fv}(T)$ as $\Gamma \vdash T\ \textbf{wf}$, by (LET), we have that $\Gamma, \Delta' \vdash \textbf{let}\, x = v\, \textbf{in}\, t$ 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.1 (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.4 (Top capture set)**.** *Let* $\Gamma \vdash C$ ***wf****. Then* $\Gamma \vdash C <: \{\mathbf{cap}\}$.

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

Corollary A.2 (Effectively top capture set)**.** *Let* $\Gamma \vdash C, D$ ***wf*** *such that* $\mathbf{cap} \in D$*. Then we can derive* $\Gamma \vdash C <: D$.

Proof. We can derive $\Gamma \vdash C <: \{\mathbf{cap}\}$ by Lemma A.4 and then we can conclude by Lemma A.7 and (SC-ELEM). □

Lemma A.5 (Universal capability subcapturing inversion)**.** *Let* $\Gamma \vdash C <: D$*. If* $\mathbf{cap} \in C$*, then* $\mathbf{cap} \in D$.

Proof. By induction on subcapturing. Case (SC-ELEM) immediate, case (SC-SET) by repeated IH, case (SC-VAR) contradictory. □

Lemma A.6 (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.7 (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.6 $\Gamma \vdash \{x\} <: C_3$.

Case (SC-VAR). Then $C_1 = \{x\}$ and $x : R \wedge C_4 \in \Gamma$ 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.8 (Subcapturing reflexivity)**.** *If* $\Gamma \vdash C$ ***wf****, then* $\Gamma \vdash C <: C$.

Proof. By (SC-SET) and (SC-ELEM). □

Lemma A.9 (Subtyping implies subcapturing)**.** *If* $\Gamma \vdash R_1\,\hat{}\,C_1 <: R_2\,\hat{}\,C_2$*, 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.7. If (REFL), then $C_1 = C_2$ and we can conclude by Lemma A.8. Otherwise, $C_1 = C_2 = \{\}$ and we can conclude by (SC-SET). □

A.3.1 Subtyping inversion

Fact 3. *Both subtyping and subcapturing are transitive.*

Proof. Subtyping is intrisically transitive through (TRANS), while subcapturing admits transitivity as per Lemma A.7. □

Fact 4. *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.8. □

Lemma A.10 (Subtyping inversion: type variable)**.** *If* $\Gamma \vdash U <: X\,\hat{}\,C$*, then* U *is of the form* $X'\,\hat{}\,C'$ *and we have* $\Gamma \vdash C' <: C$ *and* $\Gamma \vdash X' <: X$*.*

Proof. By induction on the subtyping derivation.

Case (TVAR), (REFL). Follows from reflexivity (4).

Case (CAPT). Then we have $U = S\,\hat{}\,C'$ and $\Gamma \vdash C' <: C$ and $\Gamma \vdash S <: X$.
This relationship is equivalent to $\Gamma \vdash S\,\hat{}\,\{\} <: X\,\hat{}\,\{\}$, on which we invoke the IH.
By IH we have $S\,\hat{}\,\{\} = Y\,\hat{}\,\{\}$ and we can conclude with $U = Y\,\hat{}\,C'$.

Case (TRANS). Then we have $\Gamma \vdash U <: U$ and $\Gamma \vdash U <: X\,\hat{}\,C$. We proceed by using the IH twice and conclude by transitivity (3).

Other rules are impossible.

□

Lemma A.11 (Subtyping inversion: capturing type)**.** *If* $\Gamma \vdash U <: S\,\hat{}\,C$*, then* U *is of the form* $S'\,\hat{}\,C'$ *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 (3) and reflexive (4). The result follows from straightforward induction on the subtyping derivation. □

Lemma A.12 (Subtyping inversion: function type)**.** *If $\Gamma \vdash U <: (\forall(x : T_1) T_2) \wedge C$, then U either is of the form $X \wedge C'$ and we have $\Gamma \vdash C' <: C$ and $\Gamma \vdash X <: \forall(x : T_1) T_2$, or U is of the form $(\forall(x : U_1) U_2) \wedge C'$ 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 (4).

Case (CAPT). Then we have $\Gamma \vdash C' <: C$ and $\Gamma \vdash S <: \forall(x : T_1) T_2$.
This relationship is equivalent to $\Gamma \vdash S \wedge \{\} <: (\forall(x : T_1) T_2) \wedge \{\}$, on which we invoke the IH.
By IH $S \wedge \{\}$ might have two forms. If $S \wedge \{\} = X \wedge \{\}$, then we can conclude with $U = X \wedge C'$.
Otherwise we have $S \wedge \{\} = (\forall(x : U_1) U_2) \wedge \{\}$ and $\Gamma \vdash T_1 <: U_1$ and $\Gamma, x : T_1 \vdash U_2 <: T_2$.
Then, $U = (\forall(x : U_1) U_2) \wedge C'$ lets us conclude.

Case (TRANS). Then we have $\Gamma \vdash U <: U'$ and $\Gamma \vdash U <: (\forall(x : T_1) T_2) \wedge C$. By IH U may have one of two forms. If $U = X \wedge C'$, we proceed with Lemma A.10 and conclude by transitivity (3).
Otherwise $U = (\forall(x : U_1) U_2) \wedge C'$ and we use the IH again on $\Gamma \vdash U' <: (\forall(x : U_1) U_2) \wedge C'$.
If $U = X \wedge C''$, we again can conclude by (3). Otherwise if $U = (\forall(x : U_1) U_2) \wedge C''$, 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 (3).

Other rules are not possible.

□

Lemma A.13 (Subtyping inversion: type function type)**.** *If $\Gamma \vdash U <: (\forall[X <: S] T) \wedge C$, then U either is of the form $X \wedge C'$ and we have $\Gamma \vdash C' <: C$ and $\Gamma \vdash X <: \forall[X <: S] T$, or U is of the form $(\forall[X <: R] U') \wedge C'$ and we have $\Gamma \vdash C' <: C$ and $\Gamma \vdash T <: U'$ and $\Gamma, X <: T \vdash R <: S$.*

Proof. Analogous to the proof of Lemma A.12. □

Lemma A.14 (Subtyping inversion: boxed type)**.** *If $\Gamma \vdash U <: (\square\, T) \wedge C$, then U either is of the form $X \wedge C'$ and we have $\Gamma \vdash C' <: C$ and $\Gamma \vdash X <: \square\, T$, or U is of the form $(\square\, U') \wedge C'$ and we have $\Gamma \vdash C' <: C$ and $\Gamma \vdash U' <: T$.*

Proof. Analogous to the proof of Lemma A.12. □

A.3.2 Permutation, weakening, narrowing

Lemma A.15 (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.16 (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 $\Gamma \vdash \lambda\big(y : U\big)\ t' : \forall(y : U)U$. 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 $\Gamma, x : T \vdash \lambda\big(y : U\big)\ t' : \forall(y : U)U$, which concludes.

□

Lemma A.17 (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.18 (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 $U = R \wedge C$ and $\Gamma, x : R \wedge C, \Delta \vdash x : R \wedge \{x\}$. As $\Gamma \vdash U' <: U$, from Lemma A.11 we know that $U' = R' \wedge C'$ and that $\Gamma \vdash R' <: R$. We need to prove that $\Gamma, x : R' \wedge C', \Delta \vdash x : R \wedge \{x\}$. 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.11 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.7.

□

A.4 Substitution

A.4.1 Term Substitution

We will make use of the following fact:

Fact 5. *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 $x \notin \text{fv}(T)$.*

Lemma A.19 (Term substitution preserves subcapturing)**.** *If $\Gamma, x : P, \Delta \vdash C_1 <: C_2$ and $\Gamma \vdash D <:$ $\text{cv}(P)$, 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 $y : S \wedge C_3 \in \Gamma, x : P, \Delta$ 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 $P = S \wedge C_3$ 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 $\Gamma, \theta\Delta \vdash \theta D <:$ θC_3, which means we can conclude by Lemma A.7.
Otherwise, $x \neq y$, and our goal is $\Gamma, \theta\Delta \vdash C_1 <: \theta C_2$. We inspect where y is bound.

Case $y \in \text{dom}(\Gamma)$. Then note that $y \notin C_3$ by 5. 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 $y : P \wedge C_3 \in \Gamma, \theta\Delta$.

Case $y \in \text{dom}(\Delta)$. Then $y : \theta(P \wedge C_3) \in \Gamma, \theta\Delta$ 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.7 we also have $\Gamma, \theta\Delta \vdash \{y'\} <: \theta C_2$. which is our goal.

□

Lemma A.20 (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.22 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 $x \notin \mathrm{fv}(S)$ (5), $\theta(S) = S$. 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).

$\square$

Lemma A.21 (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 $y : S \wedge C \in \Gamma, x : P, \Delta$ and $T = S \wedge \{y\}$ and our goal is $\Gamma, \theta\Delta \vdash y : \theta(S \wedge \{y\})$.
If $y = x$, then $P = S \wedge C$ and $\theta(S \wedge \{x\}) = S \wedge \{x'\}$. Our goal is $\Gamma, \theta\Delta \vdash x' : S \wedge \{x'\}$ and we can conclude by (VAR).
Otherwise, $y \neq x$ and we inspect where y is bound.
If $y \in \mathrm{dom}(\Gamma)$, then $x \notin \mathrm{fv}(S \wedge C)$ and so $\theta(S \wedge \{z\}) = S \wedge \{z\}$ and we can conclude by (VAR).
Otherwise, $y \in \mathrm{dom}(\Delta)$, so $y : \theta(S \wedge C) \in \Gamma, \theta\Delta$ and we can conclude by (VAR).

Case (SUB). By IH, Lemma A.20 and (SUB).

Case (ABS). Then $t = \lambda y Q t'$, $T = (\forall(y : Q) T') \wedge cv(t)$ 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 $\Gamma, x : P, \Delta \vdash z_1 : (\forall(y : Q) T') \wedge C$ and $\Gamma, x : P, \Delta \vdash z_1 : Q$ and $T = [y := z_2]T'$.
By IH we have $\Gamma, \theta\Delta \vdash \theta z_1 : \theta((\forall(y : Q) T') \wedge C)$ 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 $\Gamma, x : P, \Delta \vdash z : S \wedge C$ and $T = \Box S \wedge C$.
By IH, we have $\Gamma, \theta\Delta \vdash \theta z : \theta S \wedge \theta C$. 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 $t = \textbf{let}\, y = s\, \textbf{in}\, t'$ 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 $\Gamma, \theta\Delta \vdash \theta(\textbf{let}\, y = s\, \textbf{in}\, t') : \theta T$, which concludes. □

A.4.2 Type Substitution

Lemma A.22 (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 $C = \{y\}$, $y : S' \wedge C' \in \Gamma, X <: S, \Delta$, $y \neq X$. Inspect where y is bound. If $y \in \mathrm{dom}(\Gamma)$, we have $y : S' \wedge C' \in \Gamma, \theta\Delta$. Otherwise, by definition of substition we have $y : \theta S' \wedge C' \in \Gamma, \theta\Delta$. In both cases we can conclude by (SC-VAR), since y is still bound to a type whose capture set is C'.

□

Lemma A.23 (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.22 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 $X \notin \mathrm{fv}(S')$ (5), $\theta S' = S'$. 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.24 (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 $t = y$, $y : S' \wedge C \in \Gamma, X <: S, \Delta$, $y \neq X$, and our goal is $\Gamma, \theta\Delta \vdash y : \theta S' \wedge \{y\}$. Inspect where y is bound. If $y \in \mathrm{dom}(\Gamma)$, then $y : S' \wedge C \in \Gamma, \theta\Delta$ and $X \notin \mathrm{fv}(S')$ (5). Then, $\theta(S' \wedge C) = S' \wedge C$ and we can conclude by (VAR). Otherwise, $y : \theta S' \wedge C \in \Gamma, \theta\Delta$ 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 $\Gamma, X <: S, \Delta \vdash x : (\forall(z : U) T_0) \wedge C$ 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 $\Gamma, X <: S, \Delta \vdash x : (\forall[Z <: S'] T_0) \wedge C$ and $T = [Z := S'] T_0$.
We observe that $\theta[Z := S'] T_0 = [Z := \theta S']\theta T_0$. By IH, $\Gamma, \theta\Delta \vdash x : (\forall[Z <: \theta S'] T_0) \wedge C$, Then, we can conclude by (TAPP).

Case (BOX). Then $t = \Box y$ and $\Gamma, X <: S, \Delta \vdash y : S' \wedge C$ and $T = \Box S' \wedge C$, and our goal is $\Gamma, \theta\Delta \vdash y : \Box\theta(S' \wedge C)$.
Inspect where y is bound. If $y \in \mathrm{dom}(\Gamma)$, then $y : S' \wedge C \in \Gamma, \theta\Delta$ and $X \notin \mathrm{fv}(S')$ (5) . Then, $\theta(S' \wedge C') = S' \wedge C'$ and we can conclude by (BOX). Otherwise, $y : \theta S' \wedge C \in \Gamma, \theta\Delta$ 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 Lemma A.23.

Case (LET). Then $t = \textbf{let}\, y = s\, \textbf{in}\, t'$ 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 $\Gamma, \theta\Delta \vdash \theta(\textbf{let}\, y = s\, \textbf{in}\, t') : \theta T$, which concludes. □

A.5 Main Theorems – Soundness

A.5.1 Preliminaries

As we state Preservation (Theorem A.1) 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.25 (Value typing). *If $\Gamma \vdash v : T$, then T is not of the form $X \wedge C$.*

Proof. By induction on the typing derivation.
For rule (SUB), we know that $\Gamma \vdash v : U$ and $\Gamma \vdash U <: T$. Assuming $T = X \wedge C$, we have a contradiction by Lemma A.10 and IH.
Rules (BOX), (ABS), (TABS) are immediate, and other rules are not possible. □

Lemma A.26 (Canonical forms: term abstraction)**.** *If* $\Gamma \vdash v : (\forall(x : U)T) \wedge C$*, then we have* $v = \lambda\left(x : U'\right)\ t$ *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.12 and by Lemma A.25, the subtype is of the form $(\forall(y : U'')T') \wedge C'$ and we have $\Gamma \vdash U <: U''$. By IH we know that $v = \lambda\left(x : U'\right)\ t$ 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.27 (Canonical forms: type abstraction)**.** *If* $\Gamma \vdash v : (\forall[X <: S]T) \wedge C$*, then we have* $v = \lambda\left[X <: S'\right]\ t$ *and* $\Gamma \vdash S <: S'$ *and* $\Gamma, X <: S \vdash t : T$*.*

Proof. Analogous to the proof of Lemma A.26. □

Lemma A.28 (Canonical forms: boxed term)**.** *If* $\Gamma \vdash v : (\Box\, T) \wedge C$*, then* $v = \Box\, x$ *and* $\Gamma \vdash x : T$*.*

Proof. Analogous to the proof of Lemma A.26. □

Lemma A.29 (Variable typing inversion)**.** *If* $\Gamma \vdash x : S \wedge C$*, then* $x : S' \wedge C' \in \Gamma$ *and* $\Gamma \vdash S' <: S$ *and* $\Gamma \vdash \{x\} <: C$ *for some* C' *and* S'*.*

Proof. By induction on the typing derivation.

> *Case* (SUB). Then $\Gamma \vdash x : S'' \wedge C''$ and $\Gamma \vdash S'' \wedge C'' <: S \wedge C$. By the IH we have $\Gamma \vdash x : S' \wedge C'$ and $\Gamma \vdash S' <: S''$ and $\Gamma \vdash^{\wedge} x <: C''$. Then by Lemma A.11 we have $\Gamma \vdash S'' <: S$ and $\Gamma \vdash C'' <: C$, which lets us conclude by (TRANS) and transitivity of subcapturing. *Case* (VAR). Then $\Gamma \vdash x : S \wedge C'$ and $C = \{x\}$. We can conclude with $S' = S$ by (REFL) and reflexivity of subcapturing. □

Lemma A.30 (Variable lookup inversion)**.** *If we have both* $\Gamma \vdash \sigma \sim \Delta$ *and* $x : S \wedge C \in \Gamma, \Delta$*, then* $\sigma(x) = v$ *implies that* $\Gamma, \Delta \vdash v : S \wedge C$*.*

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, $U = S \wedge C$ and we can conclude by weakening. □

Lemma A.31 (Term abstraction lookup inversion). *If* $\Gamma \vdash \sigma \sim \Delta$ *and* $\Gamma, \Delta \vdash x : (\forall(z : U)\,T)\,\hat{}\,C$ *and* $\sigma(x) = \lambda\left(z : U'\right)\ t$*, then* $\Gamma, \Delta \vdash U <: U'$ *and* $\Gamma, \Delta, z : U \vdash t : T$.

Proof. A corollary of Lemma A.30 and Lemma A.26. $\square$

Lemma A.32 (Type abstraction lookup inversion). *If* $\Gamma \vdash \sigma \sim \Delta$ *and* $\Gamma, \Delta \vdash x : (\forall[Z <: U]\,T)\,\hat{}\,C$ *and* $\sigma(x) = \lambda\left[Z <: U'\right]\ t$*, then* $\Gamma, \Delta \vdash U <: U'$ *and* $\Gamma, \Delta, Z <: U \vdash t : T$.

Proof. A corollary of Lemma A.30 and Lemma A.27. $\square$

Lemma A.33 (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.30 and Lemma A.28. $\square$

A.5.2 Soundness

In this section, we show the classical soundness theorems.

Theorem A.1 (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 $t = \eta[\,x\,y\,]$ and $\sigma(x) = \lambda(z : U)\ s$ and $t' = \eta[\,[z := y]s\,]$.

By Lemma A.1, 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.31, we have $\Gamma, \Delta, z : U' \vdash s : Q''$. By Lemma A.21, 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 $t' = \eta[\,[z := y]s\,]$ can be typed as T, we use Lemma A.2.

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

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

Again, by Lemma A.1 for some Q we have $\Gamma, \Delta \vdash e : Q \Rightarrow T$ and $\Gamma, \Delta \vdash \textbf{let}\, x = y\, \textbf{in}\, s : Q$.

We again proceed by induction on number of (SUB) rules at the start of the typing derivation for $\textbf{let}\, x = y\, \textbf{in}\, s$, 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 $\Gamma, \Delta \vdash \textbf{let}\, x = y\, \textbf{in}\, s : Q'$. The premises are $\Gamma, \Delta \vdash y : U$ and $\Gamma, \Delta, x : U \vdash s : Q'$ and $x \notin \text{fv}(Q')$. By Lemma A.21, we have $\Gamma, \Delta \vdash [x := y]s : [x := y]Q'$. Because $x \notin \text{fv}(Q')$, $[x := y]Q' = Q'$, which means that we can again conclude by (SUB) and Lemma A.2.

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

Again, by Lemma A.1 for some Q we have $\Gamma, \Delta \vdash e : Q \Rightarrow T$ and $\Gamma, \Delta \vdash \textbf{let}\, x = v\, \textbf{in}\, s : Q$.

We again proceed by induction on number of (SUB) rules at the start of the typing derivation for $\textbf{let}\, x = v\, \textbf{in}\, s$, 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 $\Gamma, \Delta \vdash \textbf{let}\, x = v\, \textbf{in}\, s : Q'$. The premises are $\Gamma, \Delta \vdash v : U$ and $\Gamma, \Delta, x : U \vdash s : Q'$ and $x \notin \text{fv}(Q')$.

By weakening of typing, we also have $\Gamma, \Delta, x : U \vdash e : Q \Rightarrow T$. Then by (SUB) and Lemma A.2 we have $\Gamma, \Delta, x : U \vdash \eta[\ s\ ] : T$. Since $\Gamma, \Delta \vdash T\ \textbf{wf}$, by Barendregt $x \notin \text{fv}(T)$, so by (LET) we have $\Gamma, \Delta \vdash \textbf{let}\, x = v\, \textbf{in}\, \eta[\ s\ ] : T$, which concludes.

□

Definition A.2 (Proper configuration)**.** *We say that a term form $\sigma[\ t\ ]$ is a* canonical configuration *(of the entire term into store context σ and the plug t) if t is not of the form $\textbf{let}\, x = v\, \textbf{in}\, t'$.*

Fact 6. *Every term has a corresponding proper configuration, and finding it is decidable.*

Lemma A.34 (Extraction of bound value)**.** *If $\Gamma, \Delta \vdash x : T$ and $\Gamma \vdash \sigma \sim \Delta$ and $x \in \text{dom}(\Delta)$, then $\sigma(x) = v$.*

Proof. By structural induction on Δ. If Δ is empty, we have a contradiction. Otherwise, $\Delta = \Delta', z : T'$ and $\sigma = \sigma'[\ \textbf{let}\, z = v\, \textbf{in}[\,]\ ]$ 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.2 (Progress)**.** *If $\vdash \sigma[\ \eta[\ t\ ]\ ] : T$ and $\sigma[\ \eta[\ t\ ]\ ]$ is a proper configuration, then either $\eta[\ t\ ] = a$, or there exists $\sigma[\ t'\ ]$ such that $\sigma[\ \eta[\ t\ ]\ ] \longrightarrow \sigma[\ t'\ ]$.*

Proof. Since $\sigma[\ \eta[\ t\ ]\ ]$ is well-typed in the empty environment, there clearly must be some Δ such that $\vdash \sigma \sim \Delta$ and $\Delta \vdash \eta[\ t\ ] : T$. By Lemma A.1, 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, $\eta[\ x\ ] = \eta'[\ \textbf{let}\, y = x\, \textbf{in}\, t'\ ]$ and we can step by (RENAME); otherwise, immediate.

Case (ABS), (TABS), (BOX). Then $t = v$.
If η is non-empty, $\eta[\ v\ ] = \eta'[\ \textbf{let}\, x = v\, \textbf{in}\, t'\ ]$ and we can step by (LIFT); otherwise, immediate.

Case (APP). Then $t = x\, y$ and $\Delta \vdash x : (\forall(z : U)\, T_0)\,\hat{}\, C$ and $\Delta \vdash y : U$.
By Lemma A.34 and Lemma A.26, $\sigma(x) = \lambda\left(z : U'\right)\ t'$, which means we can step by (APPLY).

Case (TAPP). Then $t = x\,[S]$ and $\Delta \vdash x : (\forall[Z <: S]\, T_0)\,\hat{}\, C$.
By Lemma A.34 and Lemma A.27, $\sigma(x) = \lambda\left[z <: S'\right]\ t'$, which means we can step by (TAPPLY).

Case (UNBOX). Then $t = C \multimap x$ and $\Delta \vdash x : \Box S\,\hat{}\, C$.
By Lemma A.34 and Lemma A.28, $\sigma(x) = \Box\, y$, which means we can step by (OPEN).

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

Case (SUB). By IH.

□

A.5.3 Consequences

Lemma A.35 (Capture prediction for answers)**.** *If* $\Gamma \vdash \sigma[\ a\ ] : S\,\hat{}\, C$, *then* $\Gamma \vdash \sigma[\ a\ ] : S\,\hat{}\, \mathrm{cv}(\sigma[\ a\ ])$ *and* $\Gamma \vdash \mathrm{cv}(\sigma[\ a\ ]) <: C$.

Proof. By induction on the typing derivation.

Case (SUB). Then $\Gamma \vdash \sigma[\ a\ ] : S'\,\hat{}\, C'$ and $\Gamma \vdash S'\,\hat{}\, C' <: S\,\hat{}\, C$. By IH, $\Gamma \vdash \sigma[\ a\ ] : S'\,\hat{}\, \mathrm{cv}(\sigma[\ a\ ])$ and $\Gamma \vdash \mathrm{cv}(\sigma[\ a\ ]) <: C'$. By Lemma A.11, we have that $\Gamma \vdash C' <: C$ and $\Gamma \vdash S' <: S$.

To conclude we need $\Gamma \vdash \sigma[\ a\ ] : S\,\hat{}\, \mathrm{cv}(\sigma[\ a\ ])$ and $\Gamma \vdash \mathrm{cv}(\sigma[\ a\ ]) <: C$, which we respectively have by subsumption and Lemma A.7.

Case (VAR), (ABS), (TABS), (BOX). Then σ is empty and $C = \mathrm{cv}(a)$. One goal is immediate, other follows from Lemma A.8.

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

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

By Lemma A.19, we have $\Gamma \vdash [x := \mathrm{cv}(v)](\mathrm{cv}(\sigma'[\ a\ ])) <: [x := \mathrm{cv}(v)]C$.

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

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

Other rules cannot occur.

□

Lemma A.36 (Capture prediction for terms)**.** *Let* $\vdash \sigma \sim \Delta$ *and* $\Delta \vdash t : S\,\hat{}\,C$.
Then $\eta[\ t\ ] \longrightarrow^* \eta[\ \sigma'[\ a\ ]\ ]$ *implies that* $\Delta \vdash \mathrm{cv}(\sigma'[\ a\ ]) <: C$.

Proof. By preservation, $\vdash \sigma'[\ a\ ] : S\,\hat{}\,C$, which lets us conclude by Lemma A.35. □

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 $\textbf{let}\, x = v\, \textbf{in}\, \sigma'$. There are two rules that could be used to calculate $\mathrm{cv}(\textbf{let}\, x = v\, \textbf{in}\, \sigma')$:

$$\begin{aligned} \mathrm{cv}(\textbf{let}\, x = v\, \textbf{in}\, t) &= \mathrm{cv}(t) && \textbf{if}\, x \notin \mathrm{cv}(t) \\ \mathrm{cv}(\textbf{let}\, x = s\, \textbf{in}\, t) &= \mathrm{cv}(s) \cup \mathrm{cv}(t) \setminus x \end{aligned}$$

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

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

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:

$$\begin{aligned} \mathrm{resolver}(\textbf{let}\, x = v\, \textbf{in}\, \sigma) &= [x := \mathrm{cv}(v)] \circ \mathrm{resolver}(\sigma) \\ \mathrm{resolver}([\,]) &= id \end{aligned}$$

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

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

With the above, we define resolve as:

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

This definition satisfies our original desired equality with cv:

Fact 7. *For all terms t of the form $\sigma[\ s\ ]$, we have* $\text{cv}(t) = \text{resolve}(\sigma, \text{cv}(s))$

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

Accordingly, we extend cv to evaluation contexts (cv(e)) as follows:

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

We then have:

Fact 9. *For all terms t of the form $\eta[\ s\ ]$ such that s is not a value, we have* $\text{cv}(t) = \text{cv}(e) \cup \text{cv}(s)$.

A.6.3 Relating cv to store and evaluation context simultaneously

Given our definition of ‘resolve’ and cv(e), we have:

Fact 10. *Let $\sigma[\ \eta[\ t\ ]\ ]$ be a term such that t is not a value. Then:*

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

The proof proceeds by induction on σ and e, using 7 and 9.

A.6.4 Correctness of cv

Definition A.3 (Platform environment)**.** Γ *is a platform environment if for all $x \in \text{dom}(\Gamma)$ we have $x : S\,^\wedge\{\textbf{cap}\} \in \Gamma$ for some S.*

Lemma A.37 (Inversion of subcapturing under platform environment)**.** *If Γ is a platform environment and $\Gamma \vdash C <: D$, then either $C \subseteq D$ or $\textbf{cap} \in D$.*

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 $x : S\,^\wedge C' \in \Gamma$. Since Γ is a platform environment, we have $C' = \{\mathbf{cap}\}$, which means that the other premise of (SC-VAR) is $\Gamma \vdash \{\mathbf{cap}\} <: D$. Since Γ is well-formed, $\mathbf{cap} \notin \mathrm{dom}(\Gamma)$, which means that we must have $\mathbf{cap} \in D$. □

Lemma A.38 (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 $\mathbf{cap} \in D$, we trivially have the desired goal. If $\mathbf{cap} \notin D$, 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 $C = \{x\}$, $x : S \wedge C' \in \Gamma, \Gamma'$. This implies that $\Gamma = \Gamma_1, x : S \wedge C', \Gamma_2$ (as $x \notin \mathrm{dom}(\Gamma)$). Since Γ, Γ' is well-formed, we must have $\Gamma_1 \vdash C'$ **wf**. Since we already know $\mathbf{cap} \notin D$, then we must also have $\mathbf{cap} \notin C'$, 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 $x : S \wedge C' \in \Gamma$, 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.39. *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.37.

Otherwise, $\sigma = \sigma'[\,\mathbf{let}\, x = v\, \mathbf{in}[\,]\,]$ and $\Delta = \Delta', x : S_x \wedge D_x$ 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 $y : S_y \wedge D_y \in \Gamma, \Delta$ 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.30 and Lemma A.35, 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:

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

□

A.6.5 Core lemmas

Lemma A.40 (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 $t = \sigma[\ \eta[\ x\,y\ ]\ ]$ and $t' = \sigma[\ \eta[\ [z := y]s\ ]\ ]$. 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 $t = \sigma[\ \eta[\ C \multimapinv x\ ]\ ]$ and $t' = \sigma[\ \eta[\ z\ ]\ ]$. 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.30 and Lemma A.33 we have $\Gamma, \Delta \vdash z : S_z \,^\wedge C$ for some S_z. By straightforward induction on the typing derivation, we then must have $\Gamma, \Delta \vdash \{z\} <: C$. Then by Lemma A.39 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 $\mathrm{cv}(z) \subseteq \mathrm{cv}(C \multimapinv x)$ 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'.

$\Box$

Lemma A.41 (Single-step used capability 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* $\mathrm{cv}(t)$*:*

$$\{\, x \mid x \in \mathrm{used}(\Psi[\,t\,] \longrightarrow^* \Psi[\,t'\,]), x \in \mathrm{dom}(\Psi)\,\} \subseteq \mathrm{cv}(t)$$

Proof. By inspection of the reduction rule used.

Case (APPLY). Then $t = \sigma[\,\eta[\,x\,y\,]\,]$. If $x \in \mathrm{dom}(\sigma)$, the lemma trivially 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 trivially, since no capabilities are used by reducing using these rules.

$\Box$

Theorem A.3 (Used capability prediction)**.** *Let* $\Psi[\,t\,] \longrightarrow^* \Psi[\,t'\,]$*, where* $\Psi[\,t\,]$ *is a well-typed program. Then the primitive capabilities used during the reduction are a subset of the authority of t:*

$$\{\, x \mid x \in \mathrm{used}(\Psi[\,t\,] \longrightarrow^* \Psi[\,t'\,]), x \in \mathrm{dom}(\Psi)\,\} \subseteq \mathrm{cv}(t)$$

Proof. By the IH, Single-step program trace prediction and authority preservation. $\Box$

A.7 Avoidance

Here, we restate 4.3 and prove it.

Lemma A.42. *Consider a term* **let** $x = s$ **in** t *in an environment* Γ *such that* $\Gamma \vdash s : R \,\hat{}\, C_s$ *is the most specific typing for s in* Γ *and* $\Gamma, x : R \,\hat{}\, C_s \vdash t : T$ *is the most specific typing for t in the context of the body of the let, namely* $\Gamma, x : R \,\hat{}\, C_s$*. 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 U avoiding x such that* $\Gamma, x : S \,\hat{}\, C_s \vdash T <: U$*, we have* $\Gamma \vdash T' <: U$*.*

Proof. We will construct a subtyping derivation showing that $T' <: U$. Proceed by structural induction on the subtyping derivation for $T <: U$. 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 $C_T <: C_U$ for a capture set C_T from T and a capture set C_U from U, we need to show that that $\vdash [x := C_s]C_T <: C_U$. Conversely, in contravariant positions, whenever we have $C_U <: C_T$, we need to show that $C_U <: [x := \{\}]C_T$. For the covariant case, since $x \in C_T$ but not in C_U, by inverting the subcapturing relation $C_T <: C_U$, we obtain $C_s <: C_U$. Hence $[x := C_s]C_T <: C_U$, as desired.

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

B ModCC Proofs

B.1 Proof devices

We define environment well-formedness $\vdash \Gamma$ **wf** in the natural way – empty environment is well-formed; we have $\Gamma, x : T$ iff $\vdash \Gamma$ **wf** and $\Gamma \vdash T$ **wf**.

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

Fact 11. *If $\eta[\ t\ ]$ is a well-typed term in Γ, then there exist U, T such that $\Gamma \vdash \eta : U \Rightarrow T$ and finding them is decidable.*

When a module is created, our reduction rules introduce an *alias* for a region's location, e.g., if $\sigma \ni l \mapsto \textbf{mod}(l')\,\{\overline{f = r}\}$, then l' and $l.\textbf{reg}$ are aliases for the same region. Accordingly, to prove the correctness of our reduction rules we need to keep track of such aliases. We extend the syntax typing contexts Γ, Δ and allow them to contain *aliases* $p \equiv q$. We add rules for path equivalence and rules for using path equivalence during subcapturing:

Subcapturing $\boxed{\Gamma \vdash C <: C}$ **Path equivalence** $\boxed{\Gamma \vdash p \equiv q}$

SC-ALIAS
$$\frac{\Delta \vdash p \equiv q}{\Delta \vdash [x := p]C <: [x := q]C}$$

$$\frac{\Delta \ni p \equiv q}{\Delta \vdash p \equiv q} \qquad \frac{\Delta \vdash p \equiv q}{\Delta \vdash q \equiv p} \qquad \frac{\Delta \vdash p_1 \equiv p_2 \qquad \Delta \vdash p_2 \equiv p_3}{\Delta \vdash p_1 \equiv p_3}$$

Store entry typing $\boxed{\Delta \vdash l \mapsto e \sim \Delta}$

$$\frac{\overline{\Delta(r_i) \to T_i}^{\,i} \qquad \Delta' = \overline{l.f_i \equiv r_i}^{\,i}}{\Delta \vdash l \mapsto \{\overline{f_i = r_i}^{\,i}\} \sim l : \{\overline{f_i : T_i}^{\,i}\}, \Delta'}$$

$$\frac{\Delta \vdash l : \mathrm{Reg}\,\hat{}\,\{\mathbf{cap}\} \qquad \overline{\Delta(r_i) \to U_i}^{\,i} \qquad \overline{T_i = [l := x.\mathbf{reg}]U_i}^{\,i} \qquad \Delta' = l \equiv l_0.\mathbf{reg}, \overline{l.f_i \equiv r_i}^{\,i}}{\Delta \vdash l_0 \mapsto \mathbf{mod}(l)\,\{\overline{f_i = r_i}^{\,i}\} \sim l_0 : \mu x\,\{\mathbf{reg} : \mathrm{Reg}\,\hat{}\,\{\mathbf{cap}\}, \overline{f_i : T_i}^{\,i}\},}$$

$$\frac{\Delta \vdash v : T}{\Delta \vdash l_0 \mapsto v \sim l_0 : T} \qquad \Delta \vdash l \mapsto \mathbf{region}_l \sim l : \mathrm{Reg}\,\hat{}\,\{\mathbf{cap}\} \qquad \frac{\Delta \vdash v : T}{\Delta \vdash l_0 \mapsto l \rhd \mathbf{ref}\; v \sim l_0 : \mathrm{Ref}[T]\,\hat{}\,\{l\}}$$

$\sigma \sim \Delta$, where $\sigma = \overline{l_i \mapsto e_i}^{\,i}$, means:

1. $\overline{\Delta \vdash l_i \mapsto e_i \sim \Delta_i}^{\,i}$ and $\Delta = \overline{\Delta_i}^{\,i}$.
2. If e_i is a record or a module, then for some T_i we have $\Delta = \Delta', l_i : T_i, \Delta''$ and for all $r \in \mathrm{bodies}(e_i)$, we have $\Delta' \vdash r\ \mathbf{bd}$.

B.2 Properties of Evaluation Contexts and Stores

In the proof, we use the following metavariables: v, w for values, 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 $\mathrm{cv}(S\,\hat{}\,C) = C$.

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

Lemma B.1 (Evaluation context typing inversion)**.** *Let $\Gamma \vdash \eta[\,u\,] : T$. Then there exists U such that we have $\Gamma \vdash \eta : U \Rightarrow T$ and $\Gamma \vdash u : U$.*

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

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

Case (SUB). Then $\Gamma \vdash \eta[\,u\,] : T''$ and $\Gamma \vdash T'' <: T'$. We can conclude by the inner IH and (TRANS). □

Lemma B.2 (Evaluation context reification)**.** *If both* $\Gamma \vdash \eta : U \Rightarrow T$ *and* $\Gamma \vdash u : U$*, then we have* $\Gamma \vdash \eta[\, u \,] : T$*.*

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

B.3 Properties of Subcapturing

Lemma B.3 (Universal capability subcapturing inversion)**.** *Let* $\Gamma \vdash C <: D$*. If* **cap** $\in C$*, then* **cap** $\in D$*.*

Proof. By induction on subcapturing. Cases (SC-ELEM) and (SC-ALIAS) immediate, cases (SC-SET) and (SC-TRANS) by the IH, other cases contradictory. □

Lemma B.4 (Subcapturing reflexivity)**.** *If* $\Gamma \vdash C$ ***wf****, then* $\Gamma \vdash C <: C$*.*

Proof. By (SC-SET) and (SC-ELEM). □

Lemma B.5 (Subtyping implies subcapturing)**.** *If* $\Gamma \vdash S_1\,\hat{}\,C_1 <: S_2\,\hat{}\,C_2$*, then* $\Gamma \vdash C_1 <: C_2$*.*

Proof. By induction on the subtyping derivation. Case (CAPT) is immediate. Case (TRANS) follows from the IH. Case (REFL) follows from Lemma B.4. Otherwise, $C_1 = C_2 = \{\}$ and we can conclude by (SC-SET). □

B.3.1 Permutation, weakening, narrowing

Lemma B.6 (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 B.7 (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 $\Gamma \vdash \lambda\big(y : U\big)\ t' : \forall(y : U)U$. and from the premise of (ABS) we also know that $\Gamma, y : U \vdash t' : U$.

By the 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 $\Gamma, x : T \vdash \lambda\big(y : U\big)\ t' : \forall(y : U)U$, which concludes.

□

Lemma B.8 (Term binding narrowing)**.**

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

Proof. The first variant is proven by inspecting the form of p.

If $p = y$: If $y \neq x$ then y is still bound at the same type and we can conclude with $T' = T$. Otherwise, $T = U$ and $T' = U'$ and we can conclude by weakening $\Gamma \vdash U' <: U$.

If $p = q.f_0$: If $y \neq x$ then y is still bound at the same type and we can conclude with $T' = T$. Otherwise, by Lemma B.14 we know that either $U' = U$, which lets us trivially conclude, or we know that $U = \{\overline{f : U_f}^{f}\} \mathbin{\hat{}} C$ and $U' = \{\overline{f' : T'_{f'}}^{f'}\} \mathbin{\hat{}} C$ and $\Gamma \vdash \overline{T'_f <: T_f}^{f}$. Since $f_0 \in \overline{f}$, we also have $T = U_{f_0}$ and $T' = U'_{f_0}$ and $\Gamma \vdash T'_{f_0} <: T_{f_0}$. We can weaken the latter to conclude.

Other variants are proven by straightforward induction on the derivations. Only subcapturing and typing consider types to which term variables are bound. Only (SC-PATH) and (PATH) do so, which we prove below. All other cases follow from the IH or other narrowing lemmas.

Case (SC-PATH). Then $C_1 = \{p\}$ and $(\Gamma, x : U, \Delta)(p) \to S \mathbin{\hat{}} C_2$. We can conclude by lookup narrowing and (SC-TRANS).

Case (PATH). Then $t = p$ and $T = S \mathbin{\hat{}} \{p\}$ and $(\Gamma, x : U, \Delta)(p) = S \mathbin{\hat{}} C$. We can conclude by lookup narrowing and (SUB)&(CAPT).

□

B.3.2 Subtyping inversion

Fact 12. *Both subtyping and subcapturing are transitive.*

Proof. By (TRANS) and (SC-TRANS) respectively. □

Fact 13. *Both subtyping and subcapturing are reflexive.*

Proof. This is an intrinsic property of subtyping by (REFL) and an admissible property of subcapturing per Lemma B.4. □

Lemma B.9 (Subcapturing inversion: unaliased variable)**.** *Let* $\Gamma \vdash C_1 <: C_2$ *and* $\Gamma \ni x : S \mathbin{\hat{}} D$ *such that* $x \in C_1 \setminus C_2$ *and* x *has no aliases in* Γ*. Then* $\Gamma \vdash D <: C_2$*.*

Proof. By induction on the subcapturing derivation.

Case (SC-PATH). Then $C_1 = \{p\}$ and $\Gamma(x) \to R \mathbin{\hat{}} C_2$, which implies that $C_2 = D$ and lets us conclude by reflexivity (13).

Case (SC-SET). Then we have $\Gamma \vdash \{x\} <: C_2$ as a premise, and we can conclude by invoking the IH on it.

Case (SC-TRANS). Then $\Gamma \vdash C_1 <: C_3 <: C_2$. If $x \in C_3$, we can conclude by the IH on $\Gamma \vdash C_3 <: C_2$. Otherwise, we can conclude by using the IH on both premises and (SC-TRANS).

Case (SC-ALIAS), (SC-MEM), (SC-ELEM). Contradictory. □

Lemma B.10 (Subtyping inversion: capturing type)**.** *If* $\Gamma \vdash U <: S\,\hat{}\,C$*, then* U *is of the form* $S'\,\hat{}\,C'$ *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 (12) and reflexive (13). The result follows from a straightforward induction on the subtyping derivation. □

Lemma B.11 (Subtyping inversion: function type)**.** *If* $\Gamma \vdash U <: (\forall(x : T_1)\, T_2)\,\hat{}\,C$*, then* U *is of the form* $(\forall(x : U_1)\, U_2)\,\hat{}\,C'$ *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 (FUN), (REFL). Follow from reflexivity (13).

Case (CAPT). Then we have $\Gamma \vdash C' <: C$ and $\Gamma \vdash S <: \forall(x : T_1) T_2$.
This relationship is equivalent to $\Gamma \vdash S\,\hat{}\,\{\} <: (\forall(x : T_1)\, T_2)\,\hat{}\,\{\}$, on which we invoke the IH. By the IH, we have $S\,\hat{}\,\{\} = (\forall(x : U_1)\, U_2)\,\hat{}\,\{\}$ and $\Gamma \vdash T_1 <: U_1$ and $\Gamma, x : T_1 \vdash U_2 <: T_2$. Then, $U = (\forall(x : U_1)\, U_2)\,\hat{}\,C'$ lets us conclude.

Case (TRANS). Then we have $\Gamma \vdash U <: U'$ and $\Gamma \vdash U' <: (\forall(x : T_1)\, T_2)\,\hat{}\,C$. By the IH, we have $U' = (\forall(x : U'_1)\, U'_2)\,\hat{}\,C_{U'}$ and $\Gamma \vdash T_1 <: U'_1$ and $\Gamma, x : T_1 \vdash T_2 <: U'_2$. We use the IH again on $\Gamma \vdash U <: (\forall(x : U'_1)\, U'_2)\,\hat{}\,C_{U'}$. Then $U = (\forall(x : U_1)\, U_2)\,\hat{}\,C_U$ and $\Gamma \vdash U'_1 <: U_1$ and $\Gamma, x : U'_1 \vdash U_2 <: U'_2$. By narrowing we have $\Gamma, x : T_1 \vdash U_2 <: U'_2$, which lets us conclude by transitivity (12).

Other rules are not possible. □

Lemma B.12 (Subtyping inversion: boxed type)**.** *If* $\Gamma \vdash U <: \Box\, T\,\hat{}\,C$*, then* U *is of the form* $\Box\, U'\,\hat{}\,C'$ *and we have* $\Gamma \vdash C' <: C$ *and* $\Gamma \vdash U' <: T$*.*

Proof. Analogous to the proof of Lemma B.11. □

Lemma B.13 (Subtyping inversion: unit type)**.** *If* $\Gamma \vdash U <: \mathsf{Unit}\,\hat{}\,C$*, then* U *is of the form* $\mathsf{Unit}\,\hat{}\,C'$ *and we have* $\Gamma \vdash C' <: C$*.*

Proof. Analogous to the proof of Lemma B.11. □

Lemma B.14 (Subtyping inversion: record type)**.** *If* $\Gamma \vdash U <: \overline{\mu x}^{x}\, \{\overline{f_i : T_{f_i}}^{i}\}\,\hat{}\,C$ *then for some* C' *we have* $\Gamma \vdash C' <: C$ *and either*

1. U *is of the form* $\{\overline{f_j : U_{f_j}}^{j}\}\,\hat{}\,C'$ *and both* $|\overline{\mu x}^{x}| = 0$ *and* $\overline{\Gamma \vdash U_{f_i} <: T_{f_i}}^{i}$ *or*

2. $U = \overline{\mu x}^{x}\, \{\overline{f_i : T_{f_i}}^{i}\}\,\hat{}\,C$.

Proof. By induction on the subtyping derivation.

Case (REC). Then $C = \{\}$ and $|\overline{\mu x}^x| = 0$ and $U = \{\overline{f_j : U_{f_j}}^j\}$ and $\overline{\Gamma \vdash U_{f_i} <: T_{f_i}}^i$. The first conclusion holds.

Case (REFL). Then the second conclusion holds.

Case (CAPT). Then $U = S \wedge C'$ and $\Gamma \vdash C' <: C$ and $\Gamma \vdash S <: \overline{\mu x}^x \{\overline{f_i : T_{f_i}}^i\}$. We can conclude by the IH on the latter subtyping derivation, with the same conclusion as the one returned by the IH.

Case (TRANS). Then $\Gamma \vdash U <: U'$ and $\Gamma \vdash U' <: \overline{\mu x}^x \{\overline{f_i : T_{f_i}}^i\} \wedge C$. We proceed by the IH on the latter derivation.

If $U' = \{\overline{f_j : U_{f_j}}^j\} \wedge C'$: we invoke the IH on the other premise of (TRANS).

If $U = \{\overline{f_k : U_{f_k}}^k\} \wedge C''$: the second conclusion holds by subtyping transitivity.

If $U = U'$: the second conclusion holds.

If $U' = \overline{\mu x}^x \{\overline{f_i : T_{f_i}}^i\} \wedge C$: Then we can conclude by the IH on the other premise of (TRANS), with the same conclusion as the one returned by the IH. □

Lemma B.15 (Typing inversion: variable at a non-record type)**.** *Let* $\Gamma \vdash x : S \wedge C$ *such that* S *is not of the form* $\overline{\mu y}^y \{\overline{f_i : T_i}^i\}$. *Then there exist* C' *and* S' *such that* $\Gamma \ni x : S' \wedge C'$ *and* $\Gamma \vdash S' <: S$ *and also if* x *is unaliased in* Γ, *then* $\Gamma \vdash C' <: C$ *unless* $x \in C$.

Proof. By induction on the typing derivation.

Case (SUB). Then $\Gamma \vdash x : S'' \wedge C''$ and $\Gamma \vdash S'' \wedge C'' <: S \wedge C$. By the IH we have $\Gamma \vdash x : S' \wedge C'$ and $\Gamma \vdash S' <: S''$. and also if x is unaliased in Γ, then $\Gamma \vdash C' <: C''$ unless $x \in C''$. By Lemma B.10 we have $\Gamma \vdash S'' <: S$ and $\Gamma \vdash C'' <: C$. We conclude that $\Gamma \vdash S' <: S$ by (TRANS).

We proceed to proving that if x is unaliased in Γ, then $\Gamma \vdash C' <: C$ unless $x \in C$. If $x \in C$, we can immediately conclude. Otherwise, if $x \in C''$, we can conclude by Lemma B.9 on $\Gamma \vdash C'' <: C$. Otherwise, we have $\Gamma \vdash C' <: C''$, which lets us conclude by (SC-TRANS).

Case (PATH). Then $\Gamma(x) \to S \wedge C'$ and $C = \{x\}$. The former can only be derived if $\Gamma \ni x : S \wedge C'$, so we can conclude with $S' = S$ by (REFL).

Case (PACK). Impossible, since we must have $\Gamma \vdash S \wedge C$ **wf** and we assumed S is not a record type.

Case (UNPACK). We have $S = [y := x]S'$ and $\Gamma \vdash x : \mu y\, S' \wedge C$. Then $\Gamma \vdash \mu y\, S' \wedge C$ **wf**, which means that S' must be a record type, which contradicts the assumption that S is not a record type.

Other cases impossible because of the form of x. $\square$

Lemma B.16 (Typing inversion: variable at a record type)**.** *Let* $\Gamma \vdash x : \overline{\mu y}^{y}\,\{\overline{f_i : T_{f_i}}^{i}\}\,{}^\wedge C$. *Then* $\Gamma \ni x : S\,{}^\wedge C'$ *such that if* x *is unaliased in* Γ *then* $\Gamma \vdash C' <: C$ *unless* $x \in C$*, and also* $S = \overline{\mu y'}^{y'}\,\{\overline{f_j : U_{f_j}}^{j}\}$ *and:*

$$\overline{\Gamma \vdash [\overline{y' := x}^{y'}]U_{f_i} <: [\overline{y := x}^{y}]T_{f_i}}^{i}$$

Proof. By induction on the typing derivation.

Case (PATH). Then we note that $x \in C$ and conclude by reflexivity (13).

Case (PACK). Then $|\overline{\mu y}^{y}| = 1$ and $\Gamma \vdash x : ([y := x]\{\overline{f_i : T_{f_i}}^{i}\})\,{}^\wedge C$. By the IH, we have $\Gamma \ni x : \overline{\mu y'}^{y'}\,\{\overline{f_j : U_{f_j}}^{j}\}\,{}^\wedge C'$ and $\overline{\Gamma \vdash [\overline{y' := x}^{y'}]U_{f_i} <: [y := x]T_{f_i}}^{i}$, which lets us immediately conclude.

Case (UNPACK). Then $\overline{\mu y}^{y}\,\{\overline{f_i : T_{f_i}}^{i}\} = [y'' := x]\{\overline{f_i : T'_{f_i}}^{i}\}$ and $\Gamma \vdash x : \mu y''\,\{\overline{f_i : T'_{f_i}}^{i}\}\,{}^\wedge C$. By the IH we have $\Gamma \ni x : \overline{\mu y'}^{y'}\,\{\overline{f_j : U_{f_j}}^{j}\}\,{}^\wedge C'$ and $\overline{\Gamma \vdash [\overline{y' := x}^{y'}]U_{f_i} <: [y'' := x]T'_{f_i}}^{i}$. The latter type is equal to $[\overline{y' := x}^{y'}][y'' := x]T'_{f_i}$ for all i, which lets us conclude.

Case (SUB). Then $\Gamma \vdash x : U$ and $\Gamma \vdash U <: \overline{\mu y}^{y}\,\{\overline{f_i : T_{f_i}}^{i}\}\,{}^\wedge C$. By Lemma B.14, one of two cases holds:

If $U = \{\overline{f_j : U_{f_j}}^{j}\}\,{}^\wedge C'$, then $|\overline{\mu y}^{y}| = 0$ and $\Gamma \vdash C' <: C$ and $\overline{\Gamma \vdash U_{f_i} <: T_{f_i}}^{i}$. The types do not have a recursive qualifier, we can almost conclude by the IH & (TRANS) except for the subcapturing conclusion. From the IH we know that if x is unaliased in Γ, then $\Gamma \vdash C'' <: C'$ unless $x \in C'$, where $x : R\,{}^\wedge C'' \in \Gamma$ for some R. If $x \notin C'$, we can conclude by transitivity (13). Otherwise, we conclude by Lemma B.9.

If $U = \overline{\mu y}^{y}\,\{\overline{f_i : T_{f_i}}^{i}\}\,{}^\wedge C$, then we can conclude by the IH.

Other cases impossible because of the form of x. $\square$

Lemma B.17 (Typing inversion: variable)**.** *Let* $\Gamma \vdash x : \overline{\mu y}^{y}\,S\,{}^\wedge C$. *Then there exist* C' *and* S' *such that* $\Gamma \ni x : \overline{\mu y'}^{y'}\,S'\,{}^\wedge C'$ *and* $\Gamma \vdash [\overline{y := x}^{y}]S' <: [\overline{y' := x}^{y'}]S$ *and if* x *is unaliased in* Γ*, then* $\Gamma \vdash C' <: C$ *unless* $x \in C$.

Proof. If S is a record type, we can conclude by Lemma B.16 and (REC). Otherwise we can conclude by Lemma B.15. $\square$

B.4 Substitution

We will make use of the following fact:

Fact 14. *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 B.18 (Term substitution preserves lookup)**.** *If $(\Gamma, x : P, \Delta)(p) \to S \wedge C$ and $\Gamma \vdash x' : P$ and x has no aliases in $\Gamma, x : P, \Delta$, then $(\Gamma, [x := x']\Delta)([x := x']p) \to S' \wedge C'$ such that $\Gamma, [x := x']\Delta \vdash C' <: [x := x']C$ and $\Gamma, [x := x']\Delta \vdash S' <: [x := x']S$.*

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

If $p = y$ and $\Gamma, x : P, \Delta \ni y : S \wedge C$, we proceed by inspecting where y is bound.

If $y \in \mathrm{dom}(\Gamma)$: Then $\Gamma, [x := x']\Delta \ni y : S \wedge C$ and we can conclude by reflexivity (13).

If $y = x$: Then $P = S \wedge C$. By Lemma B.17 on $\Gamma \vdash x' : P$, $\Gamma \ni x' : S' \wedge C'$ and $\Gamma \vdash S' <: S$ and $\Gamma \vdash C' <: C$ unless $x' \in C$. Since we have both $x' \notin C$ (as $\Gamma, x : P, \Delta$ is well-formed) and $x' \in \mathrm{dom}(\Gamma)$, therefore we can conclude.

If $y \in \mathrm{dom}(\Delta)$: Then $\theta\Delta \ni y : \theta S \wedge \theta C$ and we can conclude by reflexivity (13).

If $p = q.f$ and $(\Gamma, x : P, \Delta)(q) \to \overline{\mu z}^{z}\{\overline{f_i : T_{f_i}}^{i}\} \wedge D$ and $S \wedge C = [\overline{z := q}^{z}]T_f$: By the IH we have $(\Gamma, \theta\Delta)(\theta q) \to U \wedge D'$ and $\Gamma, \theta\Delta \vdash D' <: \theta D$ and $\Gamma, \theta\Delta \vdash D' <: \theta(\overline{\mu z}^{z}\{\overline{f_i : T_{f_i}}^{i}\})$. By Lemma B.14, we then have either $S' = \theta(\overline{\mu z}^{z}\{\overline{f_i : T_{f_i}}^{i}\})$, or $S' = \{\overline{f_j : T'_{f_j}}^{j}\}$ and $|\overline{\mu z}^{z}| = 0$ and $\overline{\Gamma, \theta\Delta \vdash T'_{f_i} <: T_{f_i}}^{i}$. In the first case, we have $(\Gamma, \theta\Delta)(q.f) \to [\overline{z := q}^{z}]T_f$, and we can conclude by reflexivity (13). In the second case, we have $(\Gamma, \theta\Delta)(q.f) \to T'_f$, and since already we know that $\Gamma, \theta\Delta \vdash T'_f <: T_f$ we can conclude. □

Lemma B.19 (Term substitution preserves subcapturing)**.** *If $\Gamma, x : P, \Delta \vdash C_1 <: C_2$ and $\Gamma \vdash x' : P$ and x has no aliases in $\Gamma, x : P, \Delta$, then $\Gamma, [x := x']\Delta \vdash [x := x']C_1 <: [x := x']C_2$.*

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

Case (SC-PATH). Then we can conclude by Lemma B.18, the IH and (SC-TRANS).

Case (SC-ELEM). Then $C_1 = \{x\}$ and $x \in C_2$. This implies that $x' \in \theta C_2$, which lets us conclude by (SC-ELEM).

Case (SC-MEM). Then we can conclude by Lemma B.18 and the IH.

Case (SC-SET), (SC-TRANS). Then we can conclude by the IH. □

Lemma B.20 (Term substitution preserves subtyping)**.** *If $\Gamma, x : P, \Delta \vdash U <: T$ and $\Gamma \vdash x' : P$ and x has no aliases in $\Gamma, x : P, \Delta$, 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 the IH and Lemma B.19 and (CAPT).

Case (TRANS), (BOXED), (FUN), (REC). By the IH and re-application of the same rule. $\square$

Lemma B.21 (Term substitution preserves typing)**.** *If* $\Gamma, x : P, \Delta \vdash t : T$ *and* $\Gamma \vdash x' : P$. *and* x *has no aliases in* $\Gamma, x : P, \Delta$, *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 (SUB). By the IH, Lemma B.20 and (SUB).

Case (ABS). Then $t = \lambda(y : Q)\, t'$ and $T = (\forall (y : Q)\, T')\,\hat{}\,(\mathrm{cv}(t) \ominus y)$ and $\Gamma, x : P, \Delta, y : Q \vdash t' : T'$. By the IH, we have $\Gamma, \theta\Delta, y : \theta Q \vdash \theta t' : \theta T'$. Since $y \neq x$ and $y \neq x'$, we have $\theta(\mathrm{cv}(t) \ominus y) = \mathrm{cv}(\theta t) \ominus y$, which lets us conclude by (ABS).

Case (APP). Then $t = z_1\, z_2$ and $\Gamma, x : P, \Delta \vdash z_1 : (\forall (y : Q)\, T')\,\hat{}\,C$ and $\Gamma, x : P, \Delta \vdash z_1 : Q$ and $T = [y := z_2]T'$.
By the IH we have $\Gamma, \theta\Delta \vdash \theta z_1 : \theta((\forall (y : Q)\, T')\,\hat{}\,C)$ and $\Gamma, \theta\Delta \vdash \theta z_2 : \theta Q$.
Then by (APP) we have $\Gamma, \theta\Delta \vdash \theta(z_1\, z_2) : [y := \theta z_2]\theta T'$.
Since $y \neq x$ and $y \neq x'$, we have $[y := \theta z_2]\theta T' = \theta([y := z_2]T')$, which concludes.

Case (BOX). Then $t = \Box\, y$ and $\Gamma, x : P, \Delta \vdash z : S\,\hat{}\,C$ and $T = \Box\, S\,\hat{}\,C$.
By the IH, we have $\Gamma, \theta\Delta \vdash \theta y : \theta S\,\hat{}\,\theta C$. Then we can conclude by Lemma B.18 and (BOX).

Case (UNBOX). Analogous to the previous rule.

Case (LET). Then $t = \textbf{let}\; y = s\; \textbf{in}\; t'$ 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 $\Gamma, \theta\Delta \vdash \theta(\textbf{let}\; y = s\; \textbf{in}\; t') : \theta T$, which concludes.

Case (PATH). Then we can conclude by Lemma B.18 and (PATH) & (SUB) & (CAPT).

Case (PACK).
Then $t = p$ and $T = \overline{\mu y}^{y}\,(\{\overline{f_i : T_i}^{i}\})\,\hat{}\,C$ and $\Gamma, x : P, \Delta \vdash p : [y := p]\{\overline{f_i : T_i}^{i}\}\,\hat{}\,C$.
By the IH, $\Gamma, \theta\Delta \vdash \theta p : ([y := \theta p]\theta\{\overline{f_i : T_i}^{i}\})\,\hat{}\,\theta C$.
By (PACK), $\Gamma, \Delta \vdash \theta p : \mu y \theta\{\overline{f_i : T_i}^{i}\}\,\hat{}\,\theta C$, which concludes.

Case (UNPACK). Analogous to (PACK).

Case (UNIT). Immediate.

Case (MODULE). Then $t = \textbf{mod}(q)\,\{\overline{f_i = p_i}^i\}$ and $T = \mu y\,\{\overline{f_i : [q := y.\textbf{reg}]U_i}^i\}{}^\wedge\{\textbf{cap}\}$ and $\overline{\Gamma, x : P, \Delta \vdash p_i : U_i}^i$. By the IH we have $\overline{\Gamma, \theta\Delta \vdash \theta p_i : \theta U_i}^i$, and by (MODULE) we have $\Gamma, \theta\Delta \vdash \theta t = \textbf{mod}(\theta q)\,\{\overline{f_i = \theta p_i}^i\} : \mu y\,\{\overline{f_i : [\theta q := y.\textbf{reg}]\theta U_i}^i\}{}^\wedge\{\textbf{cap}\}$.
We note that $[\theta q := y.\textbf{reg}]\theta U_i = \theta[q := y.\textbf{reg}]U_i$ and we can conclude.

Case (REGION), (REF), (READ), (WRITE), (RECORD). In all these subcases we can conclude by using the IH on the premises and re-using the same typing rule. □

B.5 Main Theorems – Soundness

B.5.1 Preliminaries

Lemma B.22 (Canonical forms: term abstraction)**.** *If* $\Gamma \vdash v : (\forall(x : U)\, T){}^\wedge C$*, 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 B.11, the subtype is of the form $(\forall(y : U'')\, T){}^\wedge C''$ and we have $\Gamma \vdash U <: U''$. By the 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 B.23 (Canonical forms: boxed term)**.** *If* $\Gamma \vdash v : \Box T{}^\wedge C$*, then* $v = \Box x$ *and* $\Gamma \vdash x : U$ *and* $\Gamma \vdash U <: T$ *for some* U *such that* $\textbf{cap} \notin \mathrm{cv}(U)$*.*

Proof. Analogous to the proof of Lemma B.22. □

Lemma B.24 (Canonical forms: unit)**.** *If* $\Gamma \vdash v : \mathsf{Unit}{}^\wedge C$*, then* $v = ()$*.*

Proof. Analogous to the proof of Lemma B.22. □

Lemma B.25 (Store typing inversion: records and modules)**.** *Let* $\sigma \sim \Delta$ *and* $\Delta \ni l : \overline{\mu x}^x\,\{\overline{f_i : T_i}^i\}{}^\wedge C$*. Then* $\sigma(l)$ *is a record or a module with fields* $\overline{f_i}^i$ *whose respective bodies are* $\overline{r_i}^i$*. In addition we have* $\overline{\Delta \vdash r_i : [\overline{x := l}^x]T_i}^i$*.*

Proof. Define $U \triangleq \overline{\mu x}^x\,\{\overline{f_i : T_i}^i\}{}^\wedge C$. The store typing could only have been derived if $\sigma(l)$ is either a record or a module. If $\sigma(l)$ is a record, then we have $\Delta \vdash \sigma(l) : U$ and we can conclude by straightforward induction on this derivation. Otherwise, $\sigma(l) = \textbf{mod}(l_0)\,\{\overline{f_i = r_i}^i\}$ and $U = \mu x\{\overline{f_i : T_i}^i\}{}^\wedge C$ and we also have $\overline{\Delta \vdash r_i : T_i'}^i$ and $\overline{T_i = [l_0 := x.\textbf{reg}]\,T_i'}^i$ and $\Delta \ni l_0 \equiv l.\textbf{reg}$. Then by (SC-ALIAS) we can derive $\overline{\Delta \vdash r_i : [l_0 := l.\textbf{reg}]\,T_i'}^i$, which lets us conclude. □

Lemma B.26 (Typing inversion: stored records and modules)**.** *Let $\sigma \sim \Delta$ and $\Delta \vdash r : \overline{\mu x}^x \{\overline{f_i : T_{f_i}}^i\} \wedge C$. Then $\Delta \vdash l \mapsto e \sim \Delta'$ and $\Delta' \subseteq \Delta$ and $\sigma(r) = \sigma(l) = e$, and also $\Delta' \ni l : \overline{\mu y}^y \{\overline{f_j : T_{f_j}}^j\} \wedge C'$ and if $\Delta \vdash C <: \{\}$ then $C' = \{\}$. Finally, e is a record or a module with fields $\overline{f_i}^i$ whose respective bodies are $\overline{r_i}^i$. In addition we have $\overline{\Delta \vdash r_i : [\overline{x := r}^x]\, T_i}^i$.*

Proof. Define $U \triangleq \overline{\mu x}^x \{\overline{f_i : T_i}^i\} \wedge C$. Proceed by induction on the typing derivation.

Case (SUB). Then $\Delta \vdash r : S' \wedge C'$ and $\Delta \vdash S' \wedge C' <: U$. By Lemma B.14, either $S' \wedge C' = U$, in which case we can trivially conclude by the IH, or $|\overline{\mu x}^x| = 0$ and $\Delta \vdash C' <: C$ and $S' = \{\overline{f'_j : T'_{f'_j}}^j\}$ and $\overline{\Delta \vdash T'_{f_i} <: T_{f_i}}^i$, where $\overline{T_{f_i}}^i = \overline{T_i}^i$.

By the IH, we have $\Delta \vdash l \mapsto e \sim \Delta'$ and $\sigma(r) = \sigma(l) = e$ and $\Delta \ni l : \overline{\mu y}^y \{\overline{f_j : T_{f_j}}^j\} \wedge C''$ and if $\Delta \vdash C' <: \{\}$ then $C'' = \{\}$. We now show that if $\Delta \vdash C <: \{\}$, then $C'' = \{\}$. If it is so, then $\Delta \vdash C' <: \{\}$ by transitivity (12), which gives us $C'' = \{\}$ by the above implication, as desired.

Further by the IH, e is a record or a module with fields $\overline{f'_j}^j$ whose respective bodies are $\overline{r_{f'_j}}^j$. Since we know that $\overline{\Delta \vdash T'_{f_i} <: T_{f_i}}^i$, i.e., we can index $\overline{T'_{f'_j}}^j$ with all of $\overline{f_i}^i$, we implicitly also know that $\overline{f_i}^i \subseteq \overline{f'_j}^j$. Therefore, the IH also gives us $\overline{\Delta \vdash r_{f_i} : T'_{f_i}}^i$, where $\overline{r_{f_i}}^i = \overline{r_i}^i$. Then by (SUB) we have $\overline{\Delta \vdash r_{f_i} : T_{f_i}}^i$. Since $|\overline{\mu x}^x| = 0$, we have $\overline{T_{f_i} = [\overline{x := r}^x]\, T_{f_i}}^i$, which lets us conclude.

Case (PATH). Proceed by induction on the lookup derivation.

If $r = l$, then also $\Delta \ni l : U$. Then we can nearly conclude by Lemma B.25. We need to show that if $\Delta \vdash C <: \{\}$, then $C = \{\}$. Since we already have store entry typing, either $C = \{\textbf{cap}\}$ or $C = \{\}$; the former is contradictory and the latter is immediate.

If $r = r'.f$, then $\Delta(r') \to \overline{\mu y}^y \{\overline{f'_i : T'_{f'_i}}^i\} \wedge C'$, $f \in \overline{f'_i}^i$ and $[\overline{y := r'}^y]\, T'_f = U$.

By the IH, $\sigma(r')$ is a record or a module with fields $\overline{f'_i}^i$ whose respective bodies are $\overline{r_{f'_i}}^i$ and we have $\overline{\Delta \vdash r_{f'_i} : [\overline{y := r'}^y]\, T'_{f'_i}}^i$, as well as $\overline{\sigma(r'.f'_i) = \sigma(r_{f'_i})}^i$. Then we can conclude by the IH on $\Delta \vdash r_f : U$, since in particular $\sigma(r_f) = \sigma(r'.f)$ (and therefore by definition of store lookup, for all f' s.t. the LHS is defined we have $\sigma(r_f.f') = \sigma(r'.f.f')$).

If $\Delta \vdash r' \equiv r$, then $\Delta(r) \to U$ and $\sigma(r') = \sigma(r)$ and we can conclude by the IH. □

Lemma B.27 (Typing inversion: stored path)**.** *Let $\sigma \sim \Delta$ and $\Delta \vdash r : S \wedge C$ where S is not a record type. Then there exist l and Δ' such that $\sigma(r) = \sigma(l) = e$ and $\Delta' \subseteq \Delta$ and $\Delta \vdash l \mapsto e \sim \Delta'$ and $\Delta' \ni l : S' \wedge C'$ and $\Delta \vdash S' <: S$ and if $C = \{\}$, then $C' = \{\}$.*

Proof. We start with an induction on an ordered pair of (1) the size of the prefix of σ in which the root of r may be bound and (2) the depth of the typing derivation for r. In the base case, the root of r is the first binding in σ and the derivation depth is 1. The typing derivation could only have been derived with (PATH), and we inspect the form of r.

Case $r = l$. Then $C = \{l\}$ and $\Delta \ni l : S \wedge C'$ for some $S \wedge C'$. Then since $\sigma \sim \Delta$, we must have $\sigma(l) = e$ for some e such that $\Delta \vdash l \mapsto e \sim \Delta'$ and $\Delta' \in \Delta$. By inspecting the rules of store entry typing we must have $l : T \in \Delta'$ for some T; since $\Delta \ni l : S \wedge C'$, we have $T = S \wedge C'$. By (REFL) we have $\Delta \vdash S <: S$ and we already have $l \in C$, which lets us conclude.

Case $r = r'.f$. Then $\Delta(r') \to \overline{\mu x}^x \{\overline{f_i : T_i}^i\} \wedge C'$ and $f \in \overline{f_i}^i$. This leads to a contradiction, since by (PATH) and Lemma B.26 there must be a binding in σ that precedes the root of r'.

In the inductive case, we proceed with case analysis on the typing derivation for r.

Case (SUB). Then $\Delta \vdash r : S'' \wedge C''$ and we can conclude by the IH and transitivity of subtyping (12).

Case (PATH). As before, we inspect the form of r.

Case $r = l$. Same as before.

Case $r = r'.f$. Then as before, $\Delta(r') \to \overline{\mu x}^x \{\overline{f_i : T_{f_i}}^i\} \wedge C'$ and $f \in \overline{f_i}^i$. Then by (PATH) and Lemma B.26 $\sigma(r')$ is a record or a module with a field f whose body is r_f and we have $\Delta \vdash r_f : [\overline{x := r'}^x] T_f$. Since $\sigma \sim \Delta$, the root of r_f must be bound before the root of r, which means we can conclude by the induction hypothesis. □

Lemma B.28 (Store lookup inversion: reference's content)**.** *Let* $\sigma \sim \Delta$ *and* $\Delta \vdash r : \mathrm{Ref}[S] \wedge C$. *Then* $\sigma(r) = l \rhd \textbf{\textit{ref}}\, v$ *implies that* $\Delta \vdash v : S$.

Proof. By Lemma B.34, for some l_0 we have $\Delta \vdash l_0 \mapsto l \rhd \textbf{ref}\, v \sim l_0 : \mathrm{Ref}[S'] \wedge C'$ and $\Delta \vdash \mathrm{Ref}[S'] <: \mathrm{Ref}[S]$. The subtyping could only have been derived with (SUB), which means that $S' = S$. Therefore as a premise of store entry typing, we have $\Delta \vdash v : S$ as desired. □

Lemma B.29 (Store lookup inversion: term abstraction)**.** *If* $\sigma \sim \Delta$ *and* $\Delta \vdash r : (\forall (z : U)\ T) \wedge C$ *and* $\sigma(r) = l \rhd \lambda(z : U')\ t$*, then* $\Delta \vdash U <: U'$ *and* $\Delta, z : U \vdash t : T$.

Proof. A corollary of Lemma B.34 and Lemma B.22. □

Lemma B.30 (Store lookup inversion: box)**.** *If* $\sigma \sim \Delta$ *and* $\sigma(r) = \Box r'$ *and* $\Delta \vdash r : \Box T$*, then* $\Delta \vdash r' : U$ *and* $\Delta \vdash U <: T$ *for some* U *such that* $\mathbf{cap} \notin \mathrm{cv}(U)$.

Proof. A corollary of Lemma B.34 and Lemma B.23. □

Lemma B.31 (Store lookup inversion: untracked value)**.** *If* $\sigma \sim \Delta$ *and* $\sigma(r) = v$ *and* $\Delta \vdash r : S$*, then* $\Delta \vdash v : S$.

Proof. By Lemma B.34, for some l we have $\sigma(l) = \sigma(r) = v$ and $\Delta \ni l : R$ and $\Delta \vdash l \mapsto v \sim l : R$ and $\Delta \vdash R <: S$. Then by store entry typing we must have $\Delta \vdash v : R$ and by (SUB) & (CAPT) we can derive $\Delta \vdash v : S$, as desired. □

Lemma B.32 (Capture binding equivalence for location-rooted path aliases). *If $\sigma \sim \Delta$ and $\Delta \vdash p \equiv q$ and $\Delta(p) \to S_p \,^\wedge C_p$ and $\Delta(q) \to S_q \,^\wedge C_q$, then $\Delta \vdash C_p <: C_q$ and vice versa.*

Proof. By induction on the derivation of path equivalence. The cases for transitivity and symmetricity follow from the IH. Otherwise, $\Delta \ni p \equiv q$, in which case we know that the path equivalence was added via the store entry typing rule for a record or module. For a record, we have $C_p = C_q$ and $S_p = S_q$, which lets us conclude by reflexivity. For a module, if p is of the form $l.\mathbf{reg}$, we can conclude the same way. Otherwise, we have $C_p = [l' := l.\mathbf{reg}]C_q$, where l is the location of the module and l' is the location of the region packed with the module. Then we also have $\Delta \ni l \equiv l'.\mathbf{reg}$, which lets us conclude by (SC-ALIAS). Store typing only introduces path aliases for records and modules, and store entry typing rules for both records and modules clearly ensure the property holds. Record fields are bound at the same type as their contents and module fields have one path replaced with its alias. □

Lemma B.33 (Subcapturing inversion: location-rooted path). *Let $\sigma \sim \Delta$ and $\Delta \vdash C_1 <: C_2$. Then $r \in C_1$ and $\Delta(r) \to S \,^\wedge D$ imply that $\Delta \vdash D <: C_2$ unless C_2 contains r or its alias.*

Proof. By induction on the subcapturing derivation.

Case (SC-TRANS). Then $\Delta \vdash C_1 <: C_3 <: C_2$. We invoke the IH on $C_1 <: C_3$ and inspect its conclusion.

If $\Delta \vdash D <: C_3$: then we can conclude by (SC-TRANS).

If C_3 contains r', which is either r or its alias: Then by Lemma B.32 we have $\Delta(p) \to S' \,^\wedge D'$ such that $\Delta \vdash D <: D'$. By the IH on $C_3 <: C_2$, one of two cases holds.

If $\Delta \vdash D' <: C_2$: Then also $\Delta \vdash D <: C_2$ by transitivity (13), which concludes.

If C_2 contains r' or its alias: Then we can conclude, since r' is itself either r or its alias.

Case (SC-ALIAS). Then C_2 must contain l or its alias: either $p = l$ and q is the alias, or q is an unrelated path and C_2 contains l.

Case (SC-SET). By the IH on the $\Delta \vdash \{l\} <: D$ premise.

Case (SC-PATH), (SC-ELEM). Immediate. □

Lemma B.34 (Typing inversion: location-rooted path). *If $\sigma \sim \Delta$ and $\Delta \vdash r : S \,^\wedge C$ and $\Delta(r) \to R \,^\wedge D$, then $\Delta \vdash D <: C$ unless C contains r or its alias.*

Proof. By induction on the typing derivation.

Case (SUB). Then $\Delta \vdash r : S' \wedge C'$ and $\Delta \vdash S' \wedge C' <: S \wedge C$. By Lemma B.10 we also have $\Delta \vdash C' <: C$. By the IH one of two cases holds.

If $\Delta \vdash D <: C'$: Then we can conclude by transitivity (13).

If C' contains r or its alias: Then we can conclude by Lemma B.33.

Case (PATH). Then $D = \{r\}$, which concludes.

Case (PACK), (UNPACK). In both cases we can conclude by the IH, since in the premise r is typed at the same capture set.

Other rules cannot occur because of the form of r. □

B.5.2 Soundness

In this section, we show the classical soundness theorems.

Theorem B.1 (Preservation). *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$. Then $(\sigma, t) \longrightarrow (\sigma', t')$ implies that there exists a typing context Δ' such that $\sigma' \sim \Delta, \Delta'$ and $\Delta, \Delta' \vdash t' : T$.*

Proof. We proceed by inspecting the rule used to reduce (σ, t).

Case (APPLY). Then we have $t = \eta[\, r\, r'\,]$ and $\sigma' = \sigma$ and $t' = \eta[\, [x := r']t\,]$ and $\sigma(r) = l \rhd \lambda(x : U)\, t$.

By Lemma B.1, for some Q we have $\Delta \vdash \eta : Q \Rightarrow T$ and $\Delta \vdash r\, r' : Q$. Based on the term form, the typing derivation of $r\, r'$ must start with an arbitrary number of (SUB) rules, followed by (APP). Therefore, we proceed by structural induction on said typing derivation. In both the base and the inductive cases we can only assume that $\Delta \vdash r\, r' : Q'$ for some Q' such that $\Delta \vdash Q' <: Q$.

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

In the base case, $\Delta \vdash r\, r' : Q'$ is derived by (APP), so for some C and Q'' we have $\Delta \vdash r : (\forall(z : U')\, Q) \wedge C''$ and $\Delta \vdash r' : U'$ and $Q' = [x := r']Q''$. Since $\sigma \sim \Delta$ and $\Delta \vdash r' : U'$, clearly U' cannot be a type variable. Then by Lemma B.29, we have $\Delta, z : U' \vdash s : Q''$. By Lemma B.21, we have $\Delta \vdash [x := r']s : [x := r']Q''$, and since $Q' = [x := r']Q''$, by (SUB) we have $\Delta \vdash [x := r']s : Q$.

We pick an empty Δ' and conclude that $\Delta \vdash (t' = \eta[\, [x := r']s\,]) : T$ with Lemma B.2.

Case (OPEN). As above.

Case (GET). Then we have $t = \eta[\ !r\ ]$ and $\sigma' = \sigma$ and $t' = \eta[\ v\ ]$ and $\sigma(r) = l \triangleright \textbf{ref}\ v$.

Again, by Lemma B.1 we have $\Delta \vdash \eta : Q \Rightarrow T$ and $\Delta \vdash\ !r : Q$ As in the (APPLY) case, we proceed by induction, only working with a Q' such that $Q' <: Q$. The inductive case remains the same.

In the base case for (READ) we have $\Delta \vdash r : \text{Ref}[S] \wedge \{\textbf{cap}\}$ and $Q' = S$. By Lemma B.28 we also have $\Delta \vdash v : S$. Then we can conclude that $\Delta \vdash \eta[\ v\ ]$ with an empty Δ' with Lemma B.2.

Case (SET). Then we have $t = \eta[\ r := r'\ ]$ and $\sigma = [r \mapsto \sigma(r')]\sigma$ and $t' = \eta[\ ()\ ]$. We also implicitly know that $\sigma(r') = v$ for some v, or otherwise $[r \mapsto \sigma(r')]\sigma$ would not be a syntactically valid store.

Again, by Lemma B.1 we have $\Delta \vdash \eta : Q \Rightarrow T$ and $\Delta \vdash r := r' : Q$. As in the (APPLY) case, we proceed by induction, only working with a Q' such that $Q' <: Q$. The inductive case remains the same.

In the base case for rule (WRITE), we have $Q' = \text{Unit}$ and $\Delta \vdash l : \text{Ref}[S] \wedge \{\textbf{cap}\}$ and $\Delta \vdash y : S$. By Lemma B.15 we have $l : R \wedge D \in \Delta$ and $\Delta \vdash R <: \text{Ref}[S]$ for some D, R. Since the subtyping relation had to be derived with (REFL), in fact we have $R = \text{Ref}[S]$. Based on the $\sigma \sim \Delta$ premise, we also have $\sigma(l) = l_0 \triangleright \textbf{ref}\ w$ and $\Delta \vdash w : S$ and $\Delta \vdash \sigma(l) \sim \text{Ref}[S] \wedge D$ such that $D = \{\textbf{cap}\}$.

By Lemma B.31 we have $\Delta \vdash (\sigma(y) = v) : S$. Then we can derive $\Delta \vdash l \triangleright \textbf{ref}\ w \sim \text{Ref}[S] \wedge \{\textbf{cap}\}$, and since we already have $\sigma \sim \Delta$ this gives us $\sigma' \sim \Delta$.

To conclude, we pick an empty Δ', observe that we have $\Delta \vdash () : \text{Unit}$ and $\text{Unit} = Q'$ and $\Delta \vdash Q' <: Q$, and derive that $\Delta \vdash \eta[\ ()\ ] : T$ with (SUB) and Lemma B.2.

Case (LIFT). Then we have $t = \eta[\ v\ ]$ and $\sigma' = \sigma, l \mapsto v$ and $t' = \eta[\ l\ ]$.

Again, by Lemma B.1 for some Q we have $\Delta \vdash \eta : Q \Rightarrow T$ and $\Delta \vdash v : Q$. As in the (APPLY) case, we proceed by induction, only working with a Q' such that $Q' <: Q$. The inductive case for rule (SUB) remains the same.

In the base case, we have $\Delta \vdash v : Q'$. We inspect the form of v.

If v is not a record form: we pick $\Delta' = l : Q'$. We can derive both $\sigma' \sim \Delta, \Delta'$ and $\Delta, \Delta' \vdash \eta[\ l\ ] : T$, the latter via weakening, (SUB) and Lemma B.2, which concludes.

If v is a record form: then the last typing rule was (RECORD) and we have $v = \{\overline{f_i = r_i}^{i}\}$ and $\overline{\Delta \vdash r_i : S_i \wedge C_i}^{i}$ and $Q' = \{\overline{f_i : T_i}^{i}\} \wedge (\bigcup_i C_i)$. By Lemma B.34 we have $\overline{\Delta(r_i) \to S'_i \wedge C'_i}^{i}$ such that $\overline{\Delta \vdash S'_i <: S_i}^{i}$ and for each i we have $\Delta \vdash C'_i <: C_i$ unless C_i contains r_i or its alias.

We pick $\Delta' = \{\overline{f_i : S'_i \wedge C'_i}^i\} \wedge (\bigcup_i C_i), \overline{l_i \equiv r_i}^i$ and we derive $\sigma' \sim \Delta, \Delta'$ by extending $\sigma \sim \Delta$ via the record store entry typing rule. We can derive $\Delta, \Delta' \vdash l : Q'$ via (SUB) & (CAPT) & (REC), which leads to $\Delta, \Delta' \vdash \eta[\ l\ ] : T$ via weakening, (SUB) & Lemma B.2, which concludes.

Case (RENAME). Then we have $t = \eta[\ \textbf{let}\, x = r\, \textbf{in}\, t\ ]$ and $\sigma' = \sigma$ and $t' = \eta[\ [x := r]\, t\ ]$.

Again, by Lemma B.1 for some Q we have $\Delta \vdash \eta : Q \Rightarrow T$ and $\Delta \vdash \textbf{let}\, x = r\, \textbf{in}\, t : Q$. As in the (APPLY) case, we proceed by induction, only working with a Q' such that $Q' <: Q$. The inductive case remains the same.

In the base case, (LET) was used to derive that $\Delta \vdash \textbf{let}\, x = r\, \textbf{in}\, t : Q'$. The premises are $\Delta \vdash r : U$ and $\Delta, x : U \vdash s : Q'$ and $x \not\in \text{fv}(Q')$.
By Lemma B.21, we have $\Delta \vdash [x := r]s : [x := r]Q'$.

Because $x \not\in \text{fv}(Q')$, we also have $[x := r]Q' = Q'$, which means that we conclude with an empty Δ' by (SUB) and Lemma B.2.

Case (ALLOC). Then we have $t = \eta[\ r.\textbf{ref}\ r'\ ]$ and $\sigma' = \sigma, l \mapsto (e = l' \triangleright \textbf{ref}\, \sigma(r'))$ and $t' = \eta[\ l\ ]$, where $\sigma(r) = \textbf{region}_{l'}$

Once again, by Lemma B.1 for some Q we have $\Delta \vdash \eta : Q \Rightarrow T$ and $\Delta \vdash r.\textbf{ref}\ r' : Q$. As in the (APPLY) case, we proceed by induction, only working with a Q' such that $Q' <: Q$. The inductive case remains the same.

In the base case, (REF) was used to derive $\Delta \vdash r.\textbf{ref}\ r' : Q'$, which means we have $Q' = \text{Ref}[S] \wedge \{r\}$ and $\Delta \vdash r : \text{Reg} \wedge \{\textbf{cap}\}$ and $\Delta \vdash r' : S$.

We pick $\Delta' = l : Q'$. By Lemma B.31, we have $\Delta \vdash \sigma(r') : S$. This lets us derive $\Delta, l : Q' \vdash l \mapsto e \sim l : Q'$ by weakening, and since we already have $\sigma \sim \Delta$, this gives us $\sigma' \sim \Delta, \Delta'$. Since $\Delta \vdash Q' <: Q$, by weakening, (SUB) and Lemma B.2 we also have $\Delta, l : Q' \vdash \eta[\ l\ ] : T$, which concludes.

Case (MALLOC). Then $t = \eta[\ \textbf{mod}(r)\, \{\overline{f_i = r_i}^i\}\ ]$ and $\sigma' = \sigma, l \mapsto (e = \textbf{mod}(l')\, \{\overline{f_i = r_i}^i\})$ and $t' = \eta[\ l\ ]$, where $\sigma(r) = \textbf{region}_{l'}$.

Once again, by Lemma B.1 for some Q we have $\Delta \vdash \eta : Q \Rightarrow T$ and $\Delta \vdash \textbf{mod}(r)\, \{\overline{f_i = r_i}^i\} : Q$. As in the (APPLY) case, we proceed by induction, only working with a Q' such that $Q' <: Q$. The inductive case remains the same.

In the base case, $\Delta \vdash \textbf{mod}(r)\, \{\overline{f_i = r_i}^i\} : Q'$ was derived with (MODULE); we have $Q' = \mu x\, \{\textbf{reg} : \text{Reg} \wedge \{\textbf{cap}\}, \overline{f_i : T_i}^i\} \wedge \{\textbf{cap}\}$ and $\overline{\Delta, x : Q' \vdash r_i : U_i}^i$ and $\overline{T_i = [r := x.\textbf{reg}]U_i}^i$.

We pick $\Delta' = l : Q', l' \equiv l.\textbf{reg}, \overline{l.f_i \equiv r_i}^i$. To derive $\Delta, \Delta' \vdash l \mapsto e \sim \Delta'$ we need to show

$\overline{\Delta, \Delta' \vdash r_i : [r := l]U_i}^{\,i}$. Since we have we have $\sigma(r) = \textbf{region}_l$, either $r = l$ or r aliases l in σ. In the former case, it suffices to weaken $\overline{\Delta \vdash r_i : U_i}^{\,i}$. In the latter case, since we have $\sigma \sim \Delta$ we must also have $\Delta \vdash r \equiv l$ and we can derive $\overline{\Delta \vdash r_i : [r := l]U_i}^{\,i}$ via (SUB) & (SC-ALIAS). In either case we can derive $\Delta, \Delta' \vdash l \mapsto e \sim \Delta'$, and since we already have $\sigma \sim \Delta$ this gives us $\sigma' \sim \Delta, \Delta'$.

As before, we have $\Delta \vdash Q' <: Q$, and therefore by weakening & (SUB) & Lemma B.2 we also have $\Delta, l : Q' \vdash \eta[\, l \,] : T$, which concludes. □

Lemma B.35 (Store extraction: non-module). *Let $\sigma \sim \Delta$ and $\Delta \vdash r : S\,\hat{}\,C$. Then:*

- *If S is of the form* Reg*, then $\sigma(r) = \textbf{region}_l$*
- *If S is of the form* Ref$[S]$*, then $\sigma(r) = l \rhd \textbf{ref}\, v$*
- *If S is of the form $\forall(x : U)\, T$, then $\sigma(r) = \lambda(x : U')\, t$*
- *If S is of the form $\Box\, T$, then $\sigma(r) = \Box\, r'$*
- *If S is of the form* Unit*, then $\sigma(r) = ()$*

Proof. The first two points follow immediately from inverting the store entry typing derivation resulting from Lemma B.34.

Other points follow from inverting the store entry typing derivation resulting from Lemma B.34 and the Subtyping Inversion (Lemma B.11, Lemma B.12, Lemma B.13) and Canonical Form (Lemma B.22, Lemma B.23, Lemma B.24) Lemmas. □

Lemma B.36 (Impurity of regions). *Let $\sigma \sim \Delta$ and $\Delta \vdash l : S$. Then S is not of the form* Reg.

Proof. Assume that $S =$ Reg. By Lemma B.34, we must have $\Delta \ni l : S'\,\hat{}\,C'$ and $\Delta \vdash S' <: S$ and $\Delta \vdash C' <: \{\}$ for some type $S'\,\hat{}\,C'$. The subtyping relationship must have been derived with (REFL), which means that $S' =$ Reg. Then since $\sigma \sim \Delta$, we must have $C' = \{\textbf{cap}\}$, which is a contradiction since we cannot derive that $\Delta \vdash \{\textbf{cap}\} <: \{\}$. □

Theorem B.2 (Progress). *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$. Then either there exists r such that $t = r$, or there exist σ', t' such that $(\sigma, t) \longrightarrow (\sigma', t')$.*

Proof. Our t must be of the form $\eta[\, u \,]$ for some η, u such that u is not of the form **let** $x = u_1$ **in** u_2. By Lemma B.1, we then have $\Delta \vdash u : U$ for some U. We proceed by induction on this derivation.

Case (PATH). Then $u = r$. If $\eta \neq [\,]$, we can step by (RENAME); otherwise, $t = r$ and we can conclude.

Case (SUB). By the induction hypothesis.

Case (UNIT), (ABS), (BOX), (RECORD). Then $u = v$ and we can step by (LIFT).

Case (APP). Then $t = r\, r'$ and $\Delta \vdash r : (\forall(z : U)\, T_0)\,\hat{}\, C$ and $\Delta \vdash r' : U$. By Lemma B.35 and Lemma B.22 we have $\sigma(r) = \lambda(z : U')\, t'$, which means we can step by (APPLY).

Case (UNBOX). Then $t = C \circ\!\!- r$ and $\Delta \vdash r : (\Box S\hat{}C)\,\hat{}\, C$. By Lemma B.35 Lemma B.23, $\sigma(r) = \Box\, r'$, which means we can step by (OPEN).

Case (LET). Contradictory.

Case (REF). Then $t = r.\textbf{ref}\ r'$ and $\Delta \vdash r : \mathrm{Reg}\hat{}\{\mathbf{cap}\}$ and $\Delta \vdash r' : S$. By Lemma B.35 we have $\sigma(r) = \textbf{region}_l$. We proceed by case analysis on the form of S. If $S = \mathrm{Reg}$, then this is a contradiction by Lemma B.36. If S is a record type, then by Lemma B.26 $\sigma(r')$ must be a value—a record form—since store entry typing only assigns pure types to records. Otherwise $\sigma(r')$ is a value by Lemma B.35, which means we can step by (ALLOC).

Case (READ). Then $t = !r$ and $\Delta \vdash r : \mathrm{Ref}[S]\,\hat{}\,\{\mathbf{cap}\}$. Then by Lemma B.35 we have $\sigma(r) = l \triangleright \textbf{ref}\ v$, which means we can step by (GET).

Case (WRITE). Then $t = r := r'$ and $\Delta \vdash r : \mathrm{Ref}[S]\,\hat{}\,\{\mathbf{cap}\}$ and $\Delta \vdash r' : S$. Then by Lemma B.35 we have $\sigma(l) = l \triangleright \textbf{ref}\ v$ and by an identical argument as in the (REF) case we have $\sigma(l') = w$, which means we can step by (SET).

Case (MODULE). Then $t = \textbf{mod}(r)\,\{\overline{f_i = r_i}^{\,i}\}$ and $\Delta \vdash r : \mathrm{Reg}\hat{}\{\mathbf{cap}\}$. By Lemma B.35 we have $\sigma(r) = \textbf{region}_l$, which lets us step by (MALLOC). □

C GradCC Proofs

Figure C.1 shows all the typing rules of GradCC.

In the following proofs, we use a “logic variable” convention. A variable may be used in a position such that some instantiations of the variable don’t fit the appropriate domain; doing so implicitly narrows what the variable ranges over. For instance, if we state that $\Gamma \vdash C?_1 <: C?_2$ for some Γ and $C?_1$ and $C?_2$, then we implicitly mean that there exist C_1 and C_2 such that $C?_1 = C_1$ and $C?_2 = C_2$: the capture descriptors must be capture *sets*, since subcapturing is only defined for the latter.

C.1 Properties of Evaluation Contexts and Stores

Lemma C.1 (Evaluation context typing inversion)**.** *Let* $\Gamma \vdash \eta[\ u\ ] : T$*. Then we have* $\Gamma \vdash \eta : U \Rightarrow T$ *for some* U*, such that* $\Gamma \vdash u : U$*.*

Proof. We start exactly as in Lemma B.1, by induction on the structure of η. If $\eta = [\,]$, then $\Gamma \vdash u : T$ and clearly $\Gamma \vdash [\,] : T \Rightarrow T$. Otherwise, proceed by induction on the typing derivation of $\eta[\ u\ ]$, as before We can only assume that $\Gamma \vdash \eta[\ u\ ] : T'$ for some T' s.t. $\Gamma \vdash T' <: T$. Old cases remain the same; there is a single new case.

Case (ENCLOSURE) . Then $\eta[\ u\ ] = \mathbf{encl}[C][T']\,\eta'[\ u\ ]$. By the outer IH, we have $\Gamma \vdash \eta' : U \Rightarrow T'\,\hat{}\,D$ for some D: for all u', $\Gamma \vdash u' : U$ implies that $\Gamma \vdash \eta'[\ u'\ ] : T'$.

We want to show $\Gamma \vdash \mathbf{encl}[C][T']\,\eta' : U \Rightarrow T$, i.e., that for all u', $\Gamma \vdash u' : U$ implies that $\Gamma \vdash \eta[\ u'\ ] : T$. This goal follows from $\Gamma \vdash \eta' : U \Rightarrow T'\,\hat{}\,D$, (SUB) & (ENCL) . □

Lemma C.2 (Evaluation context reification)**.** *If both* $\Gamma \vdash \eta : U \Rightarrow T$ *and* $\Gamma \vdash u : U$*, then we have* $\Gamma \vdash \eta[\ u\ ] : T$*.*

Proof. As before, immediate from the definition of $\Gamma \vdash \eta : U \Rightarrow T$. □

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

ENCLOSURE
$$\dfrac{\Gamma \vdash t : T \qquad \Gamma \vdash C\,\mathbf{wfr}}{\Gamma \vdash \mathbf{encl}[C]\,[T]\;t : T}$$

MARK
$$\dfrac{\Gamma \vdash p : S\,\hat{}\,C?}{\Gamma \vdash \#\,p : S\,\hat{}\,\#}$$

UNIT
$$\Gamma \vdash () : \mathsf{Unit}$$

PATH
$$\dfrac{\Gamma(p) \to S\,\hat{}\;C?}{\Gamma \vdash p : S\,\hat{}\,\{p\}}$$

UNPACK
$$\dfrac{\Gamma \vdash p : \mu x\,\{\overline{f : T}\}\,\hat{}\;C?}{\Gamma \vdash p : ([x := p]\{\overline{f : T}\})\,\hat{}\;C?}$$

PACK
$$\dfrac{\Gamma \vdash p : ([x := p]\{\overline{f : T}\})\,\hat{}\;C?}{\Gamma \vdash p : \mu x\,\{\overline{f : T}\}\,\hat{}\;C?}$$

ABS
$$\dfrac{\Gamma, x : U \vdash t : T \qquad \Gamma \vdash U\,\mathbf{wf}}{\Gamma \vdash \lambda(x : U)\;t : (\forall(x : U)\;T)\,\hat{}\,(\mathrm{cv}(t) \mathbin{\dot\ominus} x)}$$

APP
$$\dfrac{\Gamma \vdash p : \forall(x : U)\;T\,\hat{}\;C? \qquad \Gamma \vdash q : U}{\Gamma \vdash p\,q : [z := q]\,T}$$

LET
$$\dfrac{\Gamma \vdash s : T \quad \Gamma, x : T \vdash t : U \quad x \notin \mathrm{fv}(U)}{\Gamma \vdash \mathbf{let}\;x = s\;\mathbf{in}\;t : U}$$

SUB
$$\dfrac{\Gamma \vdash t : T \quad \Gamma \vdash T <: U \quad \Gamma \vdash U\,\mathbf{wf}}{\Gamma \vdash t : U}$$

REGION
$$\Gamma \vdash \mathbf{region} : \mathrm{Reg}\,\hat{}\,\{\mathbf{cap}\}$$

BOX
$$\dfrac{\Gamma \vdash p : S\,\hat{}\;C? \qquad \overline{\Gamma \vdash q\;\mathbf{bd}}^{\,q \mathbin{\dot\in} C?}}{\Gamma \vdash \Box\,p : \Box\,S\,\hat{}\;C?}$$

UNBOX
$$\dfrac{\Gamma \vdash p : \Box\,S\,\hat{}\;C? \qquad \overline{\Gamma \vdash q\;\mathbf{bd}}^{\,q \mathbin{\dot\in} C?}}{\Gamma \vdash C \mathrel{\circ\!\!-} p : S\,\hat{}\,C}$$

UNBOX-MARK
$$\dfrac{\Gamma \vdash p : \Box\,S\,\hat{}\,C \qquad \overline{\Gamma \vdash q\;\mathbf{bd}}^{\,q \in C}}{\Gamma \vdash \# \mathrel{\circ\!\!-} p : S\,\hat{}\,C}$$

REF
$$\dfrac{\Gamma \vdash p : \mathrm{Reg}\,\hat{}\;C? \qquad \Gamma \vdash p : S}{\Gamma \vdash p.\mathbf{ref}\;p : S\,\hat{}\,\{p\}}$$

READ
$$\dfrac{\Gamma \vdash p : \mathrm{Ref}\,[S]\,\hat{}\;C?}{\Gamma \vdash\, !p : S}$$

WRITE
$$\dfrac{\Gamma \vdash p : \mathrm{Ref}\,[S]\,\hat{}\;C? \qquad \Gamma \vdash q : S}{\Gamma \vdash p := q : \mathsf{Unit}}$$

RECORD
$$\dfrac{\overline{\Gamma \vdash p_i : S_i\,\hat{}\;C?_i}^{\,i}}{\Gamma \vdash \{\overline{f_i = p_i}^{\,i}\} : \{\overline{f_i : S_i\,\hat{}\;C?_i}^{\,i}\}\,\hat{}\,(\dot{\bigcup}_i\;C?_i\,)}$$

MODULE
$$\dfrac{\Gamma \vdash q : \mathrm{Reg}\,\hat{}\;C? \qquad \overline{\Gamma \vdash p_i : U_i}^{\,i} \qquad \overline{T_i = [q := x.\mathbf{reg}]\,U_i}^{\,i}}{\Gamma \vdash \mathbf{mod}(q)\,\{\overline{f_i = p_i}^{\,i}\} : \mu x\,\{\mathbf{reg} : \mathrm{Reg}\,\hat{}\,\{\mathbf{cap}\}, \overline{f_i : T_i}^{\,i}\}\,\hat{}\,\{\mathbf{cap}\}}$$

Figure C.1: Full GradCC typing rules. Faded rules are unchanged. Shaded-name rules are new.

C.2 Properties of Cast Subcapturing

The previous proofs carry through without adjustments for the lemmas in this subsection, since they describe properties of subcapturing, a relation on capture *sets* and not capture descriptors.

Lemma C.3 (Universal capability subcapturing inversion)**.** *Let* $\Gamma \vdash C <: D$*. If* **cap** $\in C$*, then* **cap** $\in D$.

Proof. The same as for Lemma B.3. □

Lemma C.4 (Subcapturing reflexivity)**.** *If* $\Gamma \vdash C$ ***wf****, then* $\Gamma \vdash C <: C$.

Proof. The same as for Lemma B.4. □

Lemma C.5 (Subtyping implies subcapturing)**.** *If* $\Gamma \vdash S_1 \wedge C_1 <: S_2 \wedge C_2$*, then* $\Gamma \vdash C_1 <: C_2$.

Proof. The same as for Lemma B.5. □

C.2.1 Permutation, weakening, narrowing

Lemma C.6 (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. The order of the bindings in the environment is still not used in any rule. □

Lemma C.7 (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. The same as previously. □

Lemma C.8 (Term binding narrowing)**.**

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

Proof. The same as previously. □

C.2.2 Subtyping inversion

Fact 15. *Both subtyping and subcapturing are transitive.*

Proof. By (TRANS) and (SC-TRANS) respectively. □

Fact 16. *Both subtyping and subcapturing are reflexive.*

Proof. This is an intrinsic property of subtyping by (REFL) and an admissible property of subcapturing per Lemma B.4. □

Lemma C.9 (Subcapturing inversion: unaliased variable)**.** *Let* $\Gamma \vdash C_1 <: C_2$ *and* $\Gamma \ni x : S \wedge D$ *such that* $x \in C_1 \setminus C_2$ *and* x *has no aliases in* Γ*. Then* $\Gamma \vdash D <: C_2$.

Proof. The same as for Lemma B.9. □

Lemma C.10 (Subtyping inversion: capturing type)**.** *If* $\Gamma \vdash U <: S \wedge C?$*, then* U *is of the form* $S' \wedge C?'$ *such that either* $C?' = C? = \#$ *or* $\Gamma \vdash C?' <: C?$ *and* $\Gamma \vdash S' <: S$.

Proof. The same as for Lemma B.10, except for the new (MARKED) case which is analogous to the (CAPT) case except we conclude that $C?' = C? = \#$. □

Lemma C.11 (Subtyping inversion: function type)**.** *If* $\Gamma \vdash U <: (\forall(x : T_1)\, T_2)\,\hat{}\; C?$*, then* U *is of the form* $(\forall(x : U_1)\, U_2)\,\hat{}\; C?'$ *and we have either* $C?' = C? = \#$ *or* $\Gamma \vdash C?' <: C?$ *and* $\Gamma \vdash T_1 <: U_1$ *and* $\Gamma, x : T_1 \vdash U_2 <: T_2$*.*

Proof. The same as for Lemma B.11, with identical adjustments for the new (MARKED) case as in Lemma C.10. □

Lemma C.12 (Subtyping inversion: boxed type)**.** *If* $\Gamma \vdash U <: \Box\, T\,\hat{}\; C?$*, then* U *is of the form* $\Box\, U'\,\hat{}\; C?'$ *and we have either* $C?' = C? = \#$ *or* $\Gamma \vdash C?' <: C?$ *and* $\Gamma \vdash U' <: T$*.*

Proof. The same as for Lemma B.12, with identical adjustments for the new (MARKED) case as in Lemma C.10. □

Lemma C.13 (Subtyping inversion: unit type)**.** *If* $\Gamma \vdash U <: \mathsf{Unit}\,\hat{}\; C?$*, then* U *is of the form* $\mathsf{Unit}\,\hat{}\; C?'$ *and we have either* $C?' = C? = \#$ *or* $\Gamma \vdash C?' <: C?$*.*

Proof. The same as for Lemma B.13, with identical adjustments for the new (MARKED) case as in Lemma C.10. □

Lemma C.14 (Typing inversion: variable at a non-record type)**.** *Let* $\Gamma \vdash x : S\,\hat{}\; C?$ *such that* S *is not of the form* $\overline{\mu y}^{y}\,\{\overline{f_i : T_i}^{i}\}$*. Then there exist* $C?'$ *and* S' *such that* $\Gamma \ni x : S'\,\hat{}\; C?'$ *and* $\Gamma \vdash S' <: S$ *and also if* x *is unaliased in* Γ*, then* $\Gamma \vdash C?' <: C?$ *unless* $x \in C?$ *or* $C?' = C? = \#$*.*

Proof. The same as for Lemma B.15; the lemma statement needed to change to account for the changes in Lemma C.10. □

Lemma C.15 (Typing inversion: variable at a record type)**.** *Let* $\Gamma \vdash x : \overline{\mu y}^{y}\,\{\overline{f_i : T_{f_i}}^{i}\}\,\hat{}\; C?$*. Then* $\Gamma \ni x : S\,\hat{}\; C?'$ *such that if* x *is unaliased in* Γ*, then* $\Gamma \vdash C?' <: C?$ *unless* $x \in C?$ *or* $C?' = C? = \#$*, and also* $S = \overline{\mu y'}^{y'}\,\{\overline{f_j : U_{f_j}}^{j}\}$ *and:*

$$\overline{\Gamma \vdash [\overline{y' := x}^{y'}]\,U_{f_i} <: [\overline{y := x}^{y}]\,T_{f_i}}^{i}$$

Proof. The same as for Lemma B.16; the lemma statement needed to change to account for the changes in Lemma C.10. □

Lemma C.16 (Typing inversion: variable)**.** *Let* $\Gamma \vdash x : \overline{\mu y}^{y}\, S\,\hat{}\; C?$*. Then there exist* $C?'$ *and* S' *such that* $\Gamma \ni x : \overline{\mu y'}^{y'}\, S'\,\hat{}\; C?'$ *and* $\Gamma \vdash [\overline{y := x}^{y}]S' <: [\overline{y' := x}^{y'}]S$ *and if* x *is unaliased in* Γ*, then* $\Gamma \vdash C?' <: C?$ *unless* $x \in C?$ *or* $C?' = C? = \#$*.*

Proof. The same as for Lemma B.17; the lemma statement needed to change to account for changes in Lemma C.14 and Lemma C.15. □

C.3 Substitution

Substitution of marked paths is defined as follows:

$$
\begin{array}{rlll}
[x := p] & y & = p & \textbf{if } x = y \\
[x := p] & y & = y & \textbf{if } x \neq y \\
[x := z.\overline{f'}] & y.\overline{f} & = z.\overline{f'.f} & \textbf{if } x = y \\
[x := z.\overline{f'}] & y.\overline{f} & = y.\overline{f} & \textbf{if } x \neq y \\
[x := z.\overline{f'}] & \# y.\overline{f} & = \# z.\overline{f'.f} & \textbf{if } x = y \\
[x := z.\overline{f'}] & \# y.\overline{f} & = \# y.\overline{f} & \textbf{if } x \neq y \\
[x := \# p] & r & = [x := p]\, r & \textbf{otherwise}
\end{array}
$$

This definition naturally extends to terms just like regular substitution. Moreover, we define substition of marked paths over capture descriptors as follows:

$$
\begin{array}{rll}
[x := p] & \# & = \# \\
[x := y.\overline{f}] & \{\overline{p}\} & = \{\overline{[x := y.\overline{f}]\, p}\} \\
[x := \# p] & C? & = [x := p] C?
\end{array}
$$

Lemma C.17 (Term substitution preserves lookup)**.** *If* $(\Gamma, x : P, \Delta)(p) \rightarrow S \wedge C?$ *and* $\Gamma \vdash x' : P$ *and* x *has no aliases in* $\Gamma, x : P, \Delta$*, then* $(\Gamma, [x := x']\Delta)([x := x']p) \rightarrow S' \wedge C?'$ *such that* $\Gamma, [x := x']\Delta \vdash C?' <: [x := x']\, C?$ *and* $\Gamma, [x := x']\Delta \vdash S' <: [x := x']S$.

Proof. The same as for Lemma B.18. □

Lemma C.18 (Term substitution preserves subcapturing)**.** *If* $\Gamma, x : P, \Delta \vdash C?_1 <: C?_2$ *and* $\Gamma \vdash x' : P$ *and* x *has no aliases in* $\Gamma, x : P, \Delta$*, then* $\Gamma, [x := x']\Delta \vdash [x := x']\, C?_1 <: [x := x']\, C?_2$.

Proof. The same as for Lemma B.19. □

Lemma C.19 (Term substitution preserves subtyping)**.** *If* $\Gamma, x : P, \Delta \vdash U <: T$ *and* $\Gamma \vdash x' : P$ *and* x *has no aliases in* $\Gamma, x : P, \Delta$*, then* $\Gamma, [x := y]\Delta \vdash [x := y]U <: [x := y]T$.

Proof. The same as for Lemma B.20, except for the new (MARKED) case, which follows immediately from the IH. □

Lemma C.20 (Term substitution preserves typing)**.** *If* $\Gamma, x : P, \Delta \vdash t : T$ *and* $\Gamma \vdash x' : P$. *and* x *has no aliases in* $\Gamma, x : P, \Delta$*, then* $\Gamma, [x := x']\Delta \vdash [x := x']t : [x := x']T$.

Proof. The same as for Lemma B.21, except for three new cases.

Case (MARK) , (UNBOX-MARK) . By the IH.

Case (ENCLOSURE) . By the IH on the typing premiseand the IH on the well-formed restriction premise.

□

C.4 Main Theorems – Soundness

C.4.1 Preliminaries

Lemma C.21 (Canonical forms: term abstraction)**.** *If* $\Gamma \vdash v : (\forall(x : U)\, T)\,\hat{}\; C?$*, then we have* $v = \lambda(x : U')\, t$ *and* $\Gamma \vdash U <: U'$ *and* $\Gamma, x : U \vdash t : T$*.*

Proof. The same as for Lemma B.22. □

Lemma C.22 (Canonical forms: boxed term)**.** *If* $\Gamma \vdash v : \Box\, T\,\hat{}\; C?$*, then* $v = \Box\, x$ *and* $\Gamma \vdash x : U$ *and* $\Gamma \vdash U <: T$ *for some* U *such that* $\mathbf{cap} \notin \mathrm{cv}(U)$*.*

Proof. The same as for Lemma B.23. □

Lemma C.23 (Canonical forms: unit)**.** *If* $\Gamma \vdash v : \mathsf{Unit}\,\hat{}\; C?$*, then* $v = ()$*.*

Proof. The same as for Lemma B.24. □

Lemma C.24 (Store typing inversion: records and modules)**.** *Let* $\sigma \sim \Delta$ *and* $\Delta \ni l : \overline{\mu x}^{x}\, \{\overline{f_i : T_i}^{i}\}\,\hat{}\; C?$*. Then* $\sigma(l)$ *is a record or a module with fields* $\overline{f_i}^{i}$ *whose respective bodies are* $\overline{r_i}^{i}$*. In addition we have* $\overline{\Delta \vdash r_i : [\overline{x := l}^{x}]\, T_i}^{i}$ *.*

Proof. The same as for Lemma B.25. □

Lemma C.25 (Typing inversion: stored records and modules)**.** *Let* $\sigma \sim \Delta$ *and* $\Delta \vdash r : \overline{\mu x}^{x}\, \{\overline{f_i : T_{f_i}}^{i}\}\,\hat{}\; C?$*. Then* $\Delta \vdash l \mapsto e \sim \Delta'$ *and* $\Delta' \subseteq \Delta$ *and* $\sigma(r) = \sigma(l) = e$*, and also* $\Delta' \ni l : \overline{\mu y}^{y}\, \{\overline{f_j : T_{f_j}}^{j}\}\,\hat{}\; C?'$ *and if* $\Delta \vdash C? <: \{\}$ *then* $C?' = \{\}$*. Finally, e is a record or a module with fields* $\overline{f_i}^{i}$ *whose respective bodies are* $\overline{r_i}^{i}$*. In addition we have* $\overline{\Delta \vdash r_i : [\overline{x := r}^{x}]\, T_i}^{i}$ *.*

Proof. The same as for Lemma B.26. □

Lemma C.26 (Typing inversion: stored path)**.** *Let* $\sigma \sim \Delta$ *and* $\Delta \vdash r : S\,\hat{}\; C?$ *where* S *is not a record type. Then there exist* l *and* Δ' *such that* $\sigma(r) = \sigma(l) = e$ *and* $\Delta' \subseteq \Delta$ *and* $\Delta \vdash l \mapsto e \sim \Delta'$ *and* $\Delta' \ni l : S'\,\hat{}\; C?'$ *and* $\Delta \vdash S' <: S$ *and if* $C? = \{\}$*, then* $C?' = \{\}$*.*

Proof. The same as for Lemma B.34. □

Lemma C.27 (Store lookup inversion: reference's content)**.** *Let $\sigma \sim \Delta$ and $\Delta \vdash r : \mathrm{Ref}[S] \,\hat{}\, C?$. Then $\sigma(r) = l \rhd \textbf{ref}\, v$ implies that $\Delta \vdash v : S$.*

Proof. The same as for Lemma B.28. □

Lemma C.28 (Store lookup inversion: term abstraction)**.** *If $\sigma \sim \Delta$ and $\Delta \vdash r : (\forall(z : U)\, T) \,\hat{}\, C?$ and $\sigma(r) = l \rhd \lambda(z : U')\, t$, then $\Delta \vdash U <: U'$ and $\Delta, z : U \vdash t : T$.*

Proof. The same as for Lemma B.29. □

Lemma C.29 (Store lookup inversion: box)**.** *If $\sigma \sim \Delta$ and $\sigma(r) = \Box r'$ and $\Delta \vdash r : \Box T$, then $\Delta \vdash r' : U$ and $\Delta \vdash U <: T$ for some U such that $\mathbf{cap} \notin \mathrm{cv}(U)$.*

Proof. The same as for Lemma B.30. □

Lemma C.30 (Store lookup inversion: untracked value)**.** *If $\sigma \sim \Delta$ and $\sigma(r) = v$ and $\Delta \vdash r : S$, then $\Delta \vdash v : S$.*

Proof. The same as for Lemma B.31. □

Lemma C.31 (Capture binding equivalence for location-rooted path aliases)**.** *If $\sigma \sim \Delta$ and $\Delta \vdash p \equiv q$ and $\Delta(p) \to S_p \,\hat{}\, C?_p$ and $\Delta(q) \to S_q \,\hat{}\, C?_q$, then either ($C?_p = C?_q = \#$) or ($\Delta \vdash C?_p <: C?_q$ and vice versa).*

Proof. The same as for Lemma B.32. □

Lemma C.32 (Subcapturing inversion: location-rooted path)**.** *Let $\sigma \sim \Delta$ and $\Delta \vdash C_1 <: C_2$. Then $r \in C_1$ and $\Delta(r) \to S \,\hat{}\, D$ imply that $\Delta \vdash D <: C_2$ unless C_2 contains r or its alias.*

Proof. The same as for Lemma B.33. □

Lemma C.33 (Typing inversion: location-rooted path)**.** *If $\sigma \sim \Delta$ and $\Delta \vdash r : S \,\hat{}\, C?$ and $\Delta(r) \to R \,\hat{}\, D?$, then $\Delta \vdash D? <: D?$ unless ($C?$ contains r or its alias) or $C? = D? = \#$.*

Proof. The same as for Lemma B.34. □

C.4.2 Soundness

Theorem C.1 (Preservation). *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$. Then $(\sigma, t) \longrightarrow (\sigma', t')$ implies that there exists a typing context Δ' such that $\sigma' \sim \Delta, \Delta'$ and $\Delta, \Delta' \vdash t' : T$.*

Proof. The same as for Theorem B.1 in all the old cases. There is one new case for rule (EXIT), which can be easily concluded by nested induction (like in the other cases), the base case of which induction (for rule (ENCLOSURE)) can be concluded by (SUB) & Lemma B.2. □

Lemma C.34 (Store extraction: non-module). *Let $\sigma \sim \Delta$ and $\Delta \vdash r : S^\wedge\ C?$. Then:*

- *If S is of the form* Reg*, then $\sigma(r) = \textbf{region}_l$*
- *If S is of the form* Ref$[S]$*, then $\sigma(r) = l \rhd \textbf{ref}\, v$*
- *If S is of the form $\forall(x : U)\ T$, then $\sigma(r) = \lambda(x : U')\ t$*
- *If S is of the form $\Box\, T$, then $\sigma(r) = \Box\, r'$*
- *If S is of the form* Unit*, then $\sigma(r) = ()$*

Proof. The same as for Lemma B.35. □

Lemma C.35 (Impurity of regions). *Let $\sigma \sim \Delta$ and $\Delta \vdash l : S$. Then S is not of the form* Reg.

Proof. The same as for Lemma B.36. □

Theorem C.2 (Progress). *Let $\sigma \sim \Delta$ and $\Delta \vdash t : T$. Then either there exists r such that $t = r$, or there exist σ', t' such that $(\sigma, t) \longrightarrow (\sigma', t')$.*

Proof. Our t must be of the form $\eta[\ u\ ]$ for some η, u such that u is not of the form **let** $x = u_1$ **in** u_2 nor of the form **encl**$[C][T]\ u'$. By Lemma B.1, we then have $\Delta \vdash u : U$ for some U. We proceed as in Theorem B.2 by induction on this derivation, which has 3 new cases.

Case (MARK) . The same as the (PATH) case.

Case (UNBOX-MARK) . The same as the (UNBOX) case.

Case (ENCLOSURE) . Contradictory. □

Bibliography

Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. 2008. Semantics of Transactional Memory and Automatic Mutual Exclusion. *ACM SIGPLAN Notices* 43, 1 (Jan. 2008), 63–74. https://doi.org/10.1145/1328897.1328449 ↪ page 112

Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. Association for Computing Machinery, New York, NY, USA, 187–197. https://doi.org/10.1145/581339.581365 ↪ page 113

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: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Springer International Publishing, Cham, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14 ↪ pages 49, 95, and 96

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of Path-Dependent Types. *ACM SIGPLAN Notices* 49, 10 (Oct. 2014), 233–249. https://doi.org/10.1145/2714064.2660216 ↪ page 100

A.W. Appel. 2001. Foundational Proof-Carrying Code. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. 247–256. https://doi.org/10.1109/LICS.2001.932501 ↪ page 122

Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023a. Reference Capabilities for Flexible Memory Management. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (Oct. 2023), 270:1363–270:1393. https://doi.org/10.1145/3622846 ↪ pages 3, 113, and 117

Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023b. *Reference Capabilities for Flexible Memory Management: Extended Version.* Technical Report. https://doi.org/10.1145/3622846 arXiv:2309.02983 [cs] ↪ pages 3, 113, and 117

Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering formal metatheory. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco,* *California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 3–15. https://doi.org/10.1145/1328438.1328443 ↪ page 33

Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 84–99. https://doi.org/10.1145/3477132.3483570 ↪ page 121

Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability Types: Tracking Aliasing and Separation in Higher-Order Functional Programs. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 139:1–139:32. https://doi.org/10.1145/3485516 ↪ pages 50, 112, and 118

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 ↪ page 8

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development, Coq'Art:The Calculus of Inductive Constructions.* ↪ page 33

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2017. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 2, POPL (Dec. 2017), 8:1–8:30. https://doi.org/10.1145/3158096 ↪ page 37

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. ↪ pages 39, 45, 46, 70, and 117

Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, Jon Crowcroft and Michael Dahlin (Eds.). USENIX Association, 309–322. http://www.usenix.org/events/nsdi08/tech/full{\T1\textbackslash}_papers/bittau/bittau.pdf ↪ page 123

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 ↪ pages 34 and 63

Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, Ondřej Lhoták, and Martin Odersky. 2021. Tracking Captured Variables in Types. https://doi.org/10.48550/arXiv.2105.11896 arXiv:2105.11896 [cs] ↪ pages 4, 25, and 27

Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. 2023. Capturing Types. *ACM Transactions on Programming Languages and*

Systems 45, 4 (Nov. 2023), 21:1–21:52. https://doi.org/10.1145/3618003 ↪ pages 4, 49, 85, 105, 106, 109, and 124

Aleksander Boruch-Gruszecki, Radosław Waśko, Yichen Xu, and Lionel Parreaux. 2022. A Case for DOT: Theoretical Foundations for Objects with Pattern Matching and GADT-style Reasoning. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 179:1526–179:1555. https://doi.org/10.1145/3563342 ↪ page 100

Chandrasekhar Boyapati and Martin Rinard. 2001. A Parameterized Type System for Race-Free Java Programs. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. Association for Computing Machinery, New York, NY, USA, 56–69. https://doi.org/10.1145/504282.504287 ↪ page 112

John Boyland. 2013. Fractional Permissions. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Springer, Berlin, Heidelberg, 270–288. https://doi.org/10.1007/978-3-642-36946-9_10 ↪ page 114

John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing. In *ECOOP 2001 — Object-Oriented Programming (Lecture Notes in Computer Science)*, Jørgen Lindskov Knudsen (Ed.). Springer, Berlin, Heidelberg, 2–27. https://doi.org/10.1007/3-540-45337-7_2 ↪ pages 116 and 120

John Tang Boyland and William Retert. 2005. Connecting Effects and Uniqueness with Adoption. *ACM SIGPLAN Notices* 40, 1 (Jan. 2005), 283–295. https://doi.org/10.1145/1047659.1040329 ↪ page 114

Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as Objects in Newspeak. In *ECOOP 2010 – Object-Oriented Programming (Lecture Notes in Computer Science)*, Theo D'Hondt (Ed.). Springer, Berlin, Heidelberg, 405–428. https://doi.org/10.1007/978-3-642-14107-2_20 ↪ pages 84, 120, and 121

Jonathan Immanuel Brachthäuser and Philipp Schuster. 2017. Effekt: Extensible Algebraic Effects in Scala (Short Paper). In *Proceedings of the International Symposium on Scala* (Vancouver, BC, Canada). ACM, New York, NY, USA. https://doi.org/10.1145/3136000.3136007 ↪ page 44

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. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 76:1–76:30. https://doi.org/10.1145/3527320 ↪ pages 49, 50, 70, 108, 112, 117, 119, and 123

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 ↪ pages 2, 6, 7, 44, 50, 112, 117, 118, and 119

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 ↪ pages 2, 112, and 117

Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. 1992. Lightweight Shared Objects in a 64-Bit Operating System. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '92)*. Association for Computing Machinery, New York, NY, USA, 397–413. https://doi.org/10.1145/141936.141969 ↪ page 115

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 ↪ pages 117 and 119

Partha Das Chowdhury, Mohammad Tahaei, and Awais Rashid. 2022. Better Call Saltzer & Schroeder: A Retrospective Security Analysis of SolarWinds & Log4j. *CoRR* abs/2211.02341 (2022). https://doi.org/10.48550/arXiv.2211.02341 arXiv:2211.02341 ↪ pages 84 and 88

Chromium. 2023. Chromium sandboxing documentation. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/design/sandbox.md ↪ page 122

Dave Clarke and Sophia Drossopoulou. 2002. Ownership, Encapsulation and the Disjointness of Type and Effect. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. Association for Computing Machinery, New York, NY, USA, 292–310. https://doi.org/10.1145/582419.582447 ↪ page 113

Dave Clarke, James Noble, Tobias Wrigstad, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum (Eds.). 2013a. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification.* Lecture Notes in Computer Science, Vol. 7850. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-36946-9 ↪ pages 2, 3, and 113

Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013b. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Springer, Berlin, Heidelberg, 15–58. https://doi.org/10.1007/978-3-642-36946-9_3 ↪ pages 2, 3, and 113

Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *ECOOP 2003 – Object-Oriented Programming (Lecture Notes in Computer Science)*, Luca Cardelli (Ed.). Springer, Berlin, Heidelberg, 176–200. https://doi.org/10.1007/978-3-540-45070-2_9 ↪ page 113

David Gerard Clarke. 2002. *Object Ownership & Containment.* Thesis. UNSW Sydney. https://doi.org/10.26190/unsworks/8187 ↪ page 113

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 ↪ pages 8 and 113

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE! 2015*. ACM Press, Pittsburgh, PA, USA, 1–12. https://doi.org/10.1145/2824815.2824816 ↪ page 116

William R. Cook. 2009. On Understanding Data Abstraction, Revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 557–572. https://doi.org/10.1145/1640089.1640133 ↪ page 7

Coq. 2004. *The Coq proof assistant reference manual.* LogiCal Project. http://coq.inria.fr Version 8.0. ↪ page 33

Aaron Craig, Alex Potanin, Lindsay Groves, and Jonathan Aldrich. 2018. Capabilities: Effects for Free. In *Formal Methods and Software Engineering (Lecture Notes in Computer Science)*, Jing Sun and Meng Sun (Eds.). Springer International Publishing, Cham, 231–247. https://doi.org/10.1007/978-3-030-02450-5_14 ↪ page 5

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 ↪ page 120

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9, 3 (1966), 143–155. ↪ pages 83, 114, and 115

Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. 147–162. https://doi.org/10.1109/EuroSP.2016.22 ↪ page 116

Sophia Drossopoulou, James Noble, Mark S. Miller, and Toby Murray. 2016. Permission and Authority Revisited towards a Formalisation. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs (FTfJP'16)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2955811.2955821 ↪ page 60

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. 2006. Language Support for Fast and Reliable Ressage-Based Communication in Singularity OS. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*. Association for Computing Machinery, New York, NY, USA, 177–190. https://doi.org/10.1145/1217935.1217953 ↪ page 112

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 ↪ page 120

Matthew Flatt and Matthias Felleisen. 1998. Units: Cool Modules for HOT Languages. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 236–248. https://doi.org/10.1145/277650.277730 ↪ page 121

Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear Regions Are All You Need. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Peter Sestoft (Ed.). Springer, Berlin, Heidelberg, 7–21. https://doi.org/10.1007/11693024_2 ↪ page 42

Joseph Fourment and Yichen Xu. 2023. *A Mechanized Theory of the Box Calculus.* Technical Report. EPFL. 7 pages. https://infoscience.epfl.ch/record/302949 ↪ page 58

Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. 2021. Enclosure: Language-Based Restriction of Untrusted Libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 255–267. https://doi.org/10.1145/3445814.3446728 ↪ pages 84, 104, 109, and 123

David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming - LFP '86*. ACM Press, Cambridge, Massachusetts, United States, 28–38. https://doi.org/10.1145/319838.319848 ↪ page 111

Colin S. Gordon. 2017. A Generic Approach to Flow-Sensitive Polymorphic Effects. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 13:1–13:31. https://doi.org/10.4230/LIPIcs.ECOOP.2017.13 ↪ page 112

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 ↪ pages 5 and 50

Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Transactions on Programming Languages and Systems* 43, 1 (April 2021), 4:1–4:79. https://doi.org/10.1145/3450272 ↪ page 112

Colin S. Gordon, Matthew Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012a. Uniqueness and Reference Immutability for Safe Parallelism (Extended Version). (Oct. 2012). https://www.microsoft.com/en-us/research/publication/uniqueness-and-reference-immutability-for-safe-parallelism/ ↪ page 117

Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012b. Uniqueness and Reference Immutability for Safe Parallelism. *ACM SIGPLAN Notices* 47, 10 (Oct. 2012), 21–40. https://doi.org/10.1145/2398857.2384619 ↪ page 117

James Gosling, Bill Joy, Guy L Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification: Java SE 8 Edition.* Pearson Education. ↪ page 112

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 ↪ pages 2, 8, 42, and 111

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363 ↪ page 122

Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *ECOOP 2010 – Object-Oriented Programming (Lecture Notes in Computer Science)*, Theo D'Hondt (Ed.). Springer, Berlin, Heidelberg, 354–378. https://doi.org/10.1007/978-3-642-14107-2_17 ↪ page 113

John Hannan. 1998. A Type-Based Escape Analysis for Functional Languages. *Journal of Functional Programming* 8, 3 (May 1998), 239–273. https://doi.org/10.1017/S0956796898003025 ↪ pages 117 and 124

Norman Hardy. 1985. KeyKOS Architecture. *ACM SIGOPS Operating Systems Review* 19, 4 (Oct. 1985), 8–25. https://doi.org/10.1145/858336.858337 ↪ page 2

Norm Hardy. 1988. The Confused Deputy: (Or Why Capabilities Might Have Been Invented). *ACM SIGOPS Operating Systems Review* 22, 4 (Oct. 1988), 36–38. https://doi.org/10.1145/54289.871709 ↪ pages 88 and 115

Norman Hardy. 2023. The KeyKOS System. http://cap-lore.com/CapTheory/upenn/ ↪ page 2

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 ↪ page 15

Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 489–504. https://www.usenix.org/conference/atc19/presentation/hedayati-hodor ↪ page 123

Raphael Hiesgen, Marcin Nawrocki, Thomas C. Schmidt, and Matthias Wählisch. 2022. The Race to the Vulnerable: Measuring the Log4j Shell Incident. *CoRR* abs/2205.02544 (2022). https://doi.org/10.48550/arXiv.2205.02544 arXiv:2205.02544 ↪ page 88

Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 393–405. https://doi.org/10.1145/2976749.2978327 ↪ page 123

Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review* 41, 2 (April 2007), 37–49. https://doi.org/10.1145/1243418.1243424 ↪ page 112

Intel 2020. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel. ↪ page 123

Java. 2021. JEP 411: Deprecate the Security Manager for Removal. https://openjdk.org/jeps/411 ↪ page 86

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 41:1–41:32. https://doi.org/10.1145/3371109 ↪ pages 113 and 114

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), 66:1–66:34. https://doi.org/10.1145/3158154 ↪ page 113

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the International Conference on Functional Programming* (Boston, Massachusetts, USA). ACM, New York, NY, USA, 145–158. ↪ page 45

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In

Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09). Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596 ↪ page 2

Koka. 2023. The Koka Programming Language. https://koka-lang.github.io/koka/doc/index.html ↪ page 112

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 ↪ page 8

Ben Laurie. 2007. Safer Scripting Through Precompilation. In *Security Protocols (Lecture Notes in Computer Science)*, Bruce Christianson, Bruno Crispo, James A. Malcolm, and Michael Roe (Eds.). Springer, Berlin, Heidelberg, 289–294. https://doi.org/10.1007/978-3-540-77156-2_36 ↪ pages 108, 116, and 121

Edward Lee, Kavin Satheeskumar, and Ondřej Lhoták. 2023. Dependency-Free Capture Tracking. In *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP 2023)*. Association for Computing Machinery, New York, NY, USA, 39–43. https://doi.org/10.1145/3605156.3606454 ↪ page 108

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), 100–126. https://doi.org/10.4204/EPTCS.153.8 arXiv:1406.2061 ↪ pages 2, 70, and 112

Daan Leijen. 2016. Algebraic Effects for Functional Programming. (Aug. 2016). https://www.microsoft.com/en-us/research/publication/algebraic-effects-for-functional-programming/ ↪ pages 2 and 112

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 ↪ pages 7 and 118

Henry M. Levy. 1984. *Capability-Based Computer Systems*. Digital Press. https://homes.cs.washington.edu/{~}levy/capabook/index.html ↪ page 115

Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter R. Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 285–298. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind ↪ page 123

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 ↪ pages 7 and 118

James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 49–64. ↪ page 123

Fengyun Liu. 2016. A Study of Capability-Based Effect Systems. Master's thesis. infoscience.epfl.ch/record/219173 ↪ pages 6, 50, and 117

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 ↪ pages 2, 111, 118, and 123

Daniel Marino and Todd Millstein. 2009a. A Generic Type-and-Effect System. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/1481861.1481868 ↪ page 112

Daniel Marino and Todd D. Millstein. 2009b. 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 ↪ pages 5, 50, and 117

Daniel Marshall, Michael Vollmer, and Dominic Orchard. 2022. Linearity and Uniqueness: An Entente Cordiale. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 346–375. https://doi.org/10.1007/978-3-030-99336-8_13 ↪ pages 2 and 112

Guillaume Martres. 2023. Type-Preserving Compilation of Class-Based Languages. (Jan. 2023). https://doi.org/10.5075/epfl-thesis-8218 arXiv:2307.05557 [cs] ↪ page 96

Conor Mcbride and Philip Wadler. 2019. Doo Bee Doo Bee Doo. (2019), 54. http://homepages.inf.ed.ac.uk/slindley/papers/frankly-draft-february2019.pdf ↪ page 37

Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*. USENIX Association, 259–270. https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet ↪ page 122

Darya Melicher. 2020. *Controlling Module Authority Using Programming Language Design*. Ph. D. Dissertation. Carnegie Mellon University. ↪ pages 2, 83, 108, 109, 116, and 120

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) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 20:1–20:27. https://doi.org/10.4230/LIPIcs.ECOOP.2017.20 ↪ pages 72, 83, 84, 116, 120, and 121

Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-Language Attacks. *Proceedings 2022 Network and Distributed System Security Symposium* (2022). https://doi.org/10.14722/ndss.2022.24078 ↪ page 121

Adrian Mettler, David A. Wagner, and Tyler Close. 2010. Joe-E: A Security-Oriented Subset of Java. In *Network and Distributed System Security Symposium*, Vol. 10. 357–374. ↪ pages 108, 109, and 121

Mae Milano, Joshua Turcotti, and Andrew C. Myers. 2022. A Flexible Type System for Fearless Concurrency. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 458–473. https://doi.org/10.1145/3519939.3523443 ↪ page 3

Mark Miller. 2006. *Robust Composition: Towards a Unifed Approach to Access Control and Concurrency Control.* Ph. D. Dissertation. Johns Hopkins University. https://jscholarship.library.jhu.edu/handle/1774.2/873 ↪ pages 2, 3, 5, 60, 83, 84, 88, 90, 91, 108, 115, 116, 120, and 121

Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. *Safe Active Content in Sanitized JavaScript.* Google Inc. Technical Report. https://google-code-archive-downloads.storage.googleapis.com/v2/code.google.com/google-caja/caja-spec-2008-06-06.pdf ↪ pages 108, 116, and 121

J. C. Mitchell and R. Harper. 1988. The Essence of ML. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. Association for Computing Machinery, New York, NY, USA, 28–46. https://doi.org/10.1145/73560.73563 ↪ page 95

Eugenio Moggi. 1991. Notions of Computation and Monads. *Information and computation* 93, 1 (1991), 55–92. ↪ page 112

James H. Morris. 1973. Protection in Programming Languages. *Commun. ACM* 16, 1 (Jan. 1973), 15–21. https://doi.org/10.1145/361932.361937 ↪ pages 83, 115, and 121

Mozilla. 2023. *Firefox sandboxing documentation.* https://wiki.mozilla.org/Security/Sandbox ↪ page 122

Alan Mycroft and Janina Voigt. 2013. Notions of Aliasing and Ownership. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Springer, Berlin, Heidelberg, 59–83. https://doi.org/10.1007/978-3-642-36946-9_4 ↪ pages 2, 3, and 113

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 ↪ page 119

Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. *Retrofitting Fine Grain Isolation in the Firefox Renderer (Extended Version)*. Technical Report arXiv:2003.00572. arXiv. https://doi.org/10.48550/arXiv.2003.00572 arXiv:2003.00572 [cs] ↪ page 122

George C. Necula. 1997. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. Association for Computing Machinery, New York, NY, USA, 106–119. https://doi.org/10.1145/263699.263712 ↪ page 122

Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You Are What You Include: Large-Scale Evaluation of Remote Javascript Inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. Association for Computing Machinery, New York, NY, USA, 736–747. https://doi.org/10.1145/2382196.2382274 ↪ page 83

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 ↪ pages 8 and 113

Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2017. Simplicitly: Foundations and Applications of Implicit Function Types. *Proc. ACM Program. Lang.* 2, POPL, Article 42 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158130 ↪ page 6

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 ↪ pages 49 and 50

Martin Odersky and Guillaume Martres. 2020. Extension Methods. Scala 3 Language Reference Page. https://dotty.epfl.ch/docs/reference/contextual/extension-methods.html ↪ page 65

Peter O'Hearn. 2019. Separation Logic. *Commun. ACM* 62, 2 (Jan. 2019), 86–95. https://doi.org/10.1145/3211968 ↪ page 114

Leo Osvald, Grégory 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). ACM, New York, NY, USA, 234–251. https://doi.org/10.1145/2983990.2984009 ↪ pages 6, 8, 50, 117, 118, 119, and 123

Lionel Emile Vincent Parreaux. 2020. *Type-Safe Metaprogramming and Compilation Techniques for Designing Efficient Systems in High-Level Languages.* Ph. D. Dissertation. EPFL, Lausanne. https://doi.org/10.5075/epfl-thesis-10285 ↪ page 1

David J. Pearce. 2011. JPure: A Modular Purity System for Java. In *Compiler Construction (Lecture Notes in Computer Science)*, Jens Knoop (Ed.). Springer, Berlin, Heidelberg, 104–123. https://doi.org/10.1007/978-3-642-19861-8_7 ↪ page 112

David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Transactions on Programming Languages and Systems* 43, 1 (April 2021), 3:1–3:73. https://doi.org/10.1145/3443420 ↪ page 113

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014a. Coeffects: A Calculus of Context-Dependent Computation. *ACM SIGPLAN Notices* 49, 9 (Aug. 2014), 123–135. https://doi.org/10.1145/2692915.2628160 ↪ pages 111 and 124

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014b. 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 ↪ page 120

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/158511.158524 ↪ page 112

Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press. ↪ pages 33 and 60

Gordon Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (Feb. 2003), 69–94. https://doi.org/10.1023/A:1023064908962 ↪ pages 2, 44, and 112

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* Volume 9, Issue 4 (Dec. 2013). https://doi.org/10.2168/LMCS-9(4:23)2013 ↪ pages 2, 44, and 112

Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. 2022. Implementation Strategies for Mutable Value Semantics. *Journal of Object Technologies* 21, 2 (2022), 2–1. https://doi.org/10.5381/jot.2022.21.2.a2. ↪ page 3

Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. 2020. Kindly Bent to Free Us. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 103:1–103:29. https://doi.org/10.1145/3408985 ↪ page 114

Vineet Rajani, Deepak Garg, and Tamara Rezk. 2016. On Access Control, Capabilities, Their Equivalence, and Confused Deputy Attacks. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 150–163. https://doi.org/10.1109/CSF.2016.18 ↪ pages 88 and 115

Marianna Rapoport and Ondřej Lhoták. 2019. A Path To DOT: Formalizing Fully-Path-Dependent Types. *arXiv:1904.07298 [cs]* (April 2019). arXiv:1904.07298 [cs] http://arxiv.org/abs/1904.07298 ↪ pages 96 and 100

Jonathan A. Rees. 1996. *A Security Kernel Based on the Lambda-Calculus.* Technical Report. https://dspace.mit.edu/handle/1721.1/5944 ↪ pages 83, 115, and 121

J.C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817 ↪ page 114

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 ↪ page 49

Rust. 2023. The Rust Programming Language. https://www.rust-lang.org/ ↪ pages 2, 3, and 113

Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *ECOOP 2012 – Object-Oriented Programming (Lecture Notes in Computer Science)*, James Noble (Ed.). Springer, Berlin, Heidelberg, 258–282. https://doi.org/10.1007/978-3-642-31057-7_13 ↪ pages 112, 118, and 124

Amr Sabry and Matthias Felleisen. 1993. Reasoning about Programs in Continuation-Passing Style. In *LISP AND SYMBOLIC COMPUTATION*. 288–298. ↪ page 15

Jerome H. Saltzer. 1974. Protection and the Control of Information Sharing in Multics. *Commun. ACM* 17, 7 (July 1974), 388–402. https://doi.org/10.1145/361011.361067 ↪ pages 83 and 115

Sandboxdb. 2023. Sandboxdb.org. https://sandboxdb.org ↪ page 86

Scala. 2022a. Scala 3 API: scala.util.boundary. https://www.scala-lang.org/api/3.3.0/scala/util/boundary\protect\T1\textdollar.html ↪ page 69

Scala. 2022b. Scala 3: Capture Checking. https://dotty.epfl.ch/docs/reference/experimental/cc.html ↪ pages 9, 51, and 65

Scala. 2022c. The Scala 3 compiler, also known as Dotty. https://dotty.epfl.ch ↪ page 51

ScalaXML. 2023. Scala XML: the standard Scala XML library. https://github.com/scala/scala-xml ↪ page 107

Gabriel Scherer and Jan Hoffmann. 2013. Tracking Data-Flow with Open Closure Types. In *Logic for Programming, Artificial Intelligence, and Reasoning (Lecture Notes in Computer Science)*, Ken McMillan, Aart Middeldorp, and Andrei Voronkov (Eds.). Springer, Berlin, Heidelberg, 710–726. https://doi.org/10.1007/978-3-642-45221-5_47 ↪ pages 117 and 124

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*. https://www.semanticscholar.org/paper/Gradual-Typing-for-Functional-Languages-Siek/b7ca4b0e6d3119aa341af73964dbe38d341061dd ↪ pages 84 and 101

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 ↪ page 119

Fred Spiessens and Peter Van Roy. 2005. The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language. In *Multiparadigm Programming in Mozart/Oz (Lecture Notes in Computer Science)*, Peter Van Roy (Ed.). Springer, Berlin, Heidelberg, 21–40. https://doi.org/10.1007/978-3-540-31845-3_3 ↪ pages 108, 116, and 121

Marc Stiegler. 2007. Emily: A High Performance Language for Enabling Secure Cooperation. In *Fifth International Conference on Creating, Connecting and Collaborating through Computing (C5 '07)*. 163–169. https://doi.org/10.1109/C5.2007.13 ↪ pages 108 and 116

Marc Stiegler and Mark Miller. 2006. *How Emily Tamed the Caml.* Hewlett Packard Labs Tech Report. https://www.hpl.hp.com/techreports/2006/HPL-2006-116.pdf ↪ pages 108, 116, and 121

Nicolas Alexander Stucki. 2023. *Scalable Metaprogramming in Scala 3.* Ph. D. Dissertation. EPFL, Lausanne. https://doi.org/10.5075/epfl-thesis-8257 ↪ page 1

Jean-Pierre Talpin and Pierre Jouvelot. 1992. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming* 2, 3 (July 1992), 245–271. https://doi.org/10.1017/S0956796800000393 ↪ page 111

Ross Tate. 2013. The Sequential Semantics of Producer Effect Systems. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2429069.2429074 ↪ page 112

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (Feb. 1997), 109–176. https://doi.org/10.1006/inco.1996.2613 ↪ pages 8, 41, 111, 118, and 123

R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. 2005. Intel Virtualization Technology. *Computer* 38, 5 (May 2005), 48–56. https://doi.org/10.1109/MC.2005.163 ↪ page 123

Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner ↪ page 123

Verse. 2023. The Verse Language Reference. https://dev.epicgames.com/documentation/en-us/uefn/verse-language-reference ↪ page 2

Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*, Vol. 3. 5. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=24c850390fba27fc6f3241cb34ce7bc6f3765627 ↪ pages 2, 8, 112, and 114

Philip Wadler. 2015. A Complement to Blame. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 309–320. https://doi.org/10.4230/LIPIcs.SNAPL.2015.309 ↪ pages 84 and 101

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Giuseppe Castagna (Ed.). Springer, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-00590-9_1 ↪ pages 84 and 101

Philip Wadler and Peter Thiemann. 2003. The Marriage of Effects and Monads. *ACM Transactions on Computational Logic* 4, 1 (Jan. 2003), 1–32. https://doi.org/10.1145/601775.601776 ↪ pages 2 and 112

WASI. 2023. Webassembly: WASI. https://github.com/WebAssembly/WASI ↪ page 122

WASM-JS. 2023. Webassembly: JavaScript API. https://webassembly.github.io/spec/js-api/index.html ↪ page 122

WASM-Web. 2023. Webassembly: Web API. https://webassembly.github.io/spec/web-api/index.html ↪ page 122

Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 304–316. https://doi.org/10.1145/605397.605429 ↪ page 123

Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 457–468. https://doi.org/10.1109/ISCA.2014.6853201 ↪ page 123

A. K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (Nov. 1994), 38–94. https://doi.org/10.1006/inco.1994.1093 ↪ page 99

Yichen Xu and Martin Odersky. 2023. Formalizing Box Inference for Capture Calculus. https://doi.org/10.48550/arXiv.2306.06496 arXiv:2306.06496 [cs] ↪ page 68

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted X86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*. 79–93. https://doi.org/10.1109/SP.2009.25 ↪ page 122

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-Safe Effect Handlers via Tunneling. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 5:1–5:29. https://doi.org/10.1145/3290318 ↪ pages 37, 44, and 117

Aleksander Boruch-Gruszecki | CV

Rue de Verdeaux 17b – 1020 Renens VD – Switzerland
+41 78 233 05 75 • aleksander.boruch-gruszecki@epfl.ch
abgruszecki

Education

EPFL (Swiss Federal Institute of Technology) **Lausanne**
Pursuing a Ph.D. in Computer Science *2018–Present*

Wrocław University of Science and Technology **Wrocław**
M.Sc. in Computer Science *2012–2017*

Languages: English (bilingual) Polish (mother tongue)

Research interests: Programming languages and type systems. I am particularly interested in developing formal foundations for PL features which are intuitive to the users.

Publications

- **Aleksander Boruch-Gruszecki**, Radosław Waśko, Yichen Xu, Lionel Parreaux. 2022. **A case for DOT: theoretical foundations for object-oriented pattern matching and GADT-style reasoning**. Proceedings of the ACM on Programming Languages 6 (**OOPSLA2 2022**), 1526-1555.
(DOI: https://doi.org/10.1145/3563342)
I have significantly contributed to the writing. I am the main author of the described implementation. I have supervised the projects of Radosław Waśko and Yichen Xu included as part of this paper, and I have been the driving force behind organizing our work into a publication.
- Martin Odersky, **Aleksander Boruch-Gruszecki**, Jonathan Immanuel Brachthäuser, Edward Lee, Ondřej Lhoták. 2021. **Safer Exceptions for Scala**. Proceedings of the 12th ACM SIGPLAN International Symposium on Scala (**SCALA 2021**), 1-11. (DOI: https://doi.org/10.1145/3486610.3486893)
I have significantly contributed to the writing and the design of the presented formal system.
- **Aleksander Boruch-Gruszecki**, Jonathan Immanuel Brachthäuser, Edward Lee, Ondřej Lhoták, Martin Odersky. 2021. **Tracking Captured Variables in Types**. **arXiv 2021** preprint.
(DOI: https://doi.org/10.48550/arXiv.2105.11896)
I have significantly contributed to the writing, the design of the presented formal system and the mechanized proof of soundness.
- Martin Odersky, **Aleksander Boruch-Gruszecki**, Edward Lee, Jonathan Immanuel Brachthäuser, Ondřej Lhoták. 2022. **Scoped Capabilities for Polymorphic Effects**. **arXiv 2022** preprint.
(DOI: https://doi.org/10.48550/arXiv.2207.03402).
I have significantly contributed to the writing, the design of the presented formal system. I am the author of the attached proof of soudness.
- Lionel Parreaux, **Aleksander Boruch-Gruszecki**, Paolo G. Giarrusso. 2019. **Towards Improved GADT Reasoning in Scala**. Proceedings of the Tenth ACM SIGPLAN Symposium on Scala (**SCALA 2019**), 12-16. (DOI: https://doi.org/10.1145/3337932.3338813)
The paper discusses an approach independently discovered by me and Lionel Parreaux. I have significantly contributed to the writing.

Presentations, Invitations

- (2022) *Effects, Capabilities and Boxes.* Conference talk, OOPSLA.
- (2022) *A case for DOT.* Conference talk, OOPSLA.
- (2021) *Safer Exceptions for Scala.* Conference talk, SCALA.
- (2018, 2019) Google Compiler Summit, Munich.
- (2018) *GADTs in Scala.* Industry talk, Typelevel Summit, Lausannne.

Experience

Teaching Assistanship

CS-452: Foundations of Software (2020, 2021, 2022)
CS-206: Parallelism and Concurency (2020)
CS-210: Functional Programming (2019)
MATH-106(e): Analysis II (2019)

Industry

Bright IT **Wrocław**
Scala Software Developer *2014–2018*

- Developed a Scala database-querying library.
- Developed an XSLT-like DSL for transforming XML with Scala.
- Developed a JS library for web form scripting/validation that runs both in the browser and on the JVM.

2/2