Degrees of Separation: A Flexible Type System for Safe Concurrency

YICHEN XU, EPFL, Switzerland
ALEKSANDER BORUCH-GRUSZECKI, EPFL, Switzerland
MARTIN ODERSKY, EPFL, Switzerland

Data races have long been a notorious problem in concurrent programming. They are hard to detect, and lead to non-deterministic behaviours. There has been a lot of interest in type systems that statically guarantee data race freedom. Significant progress has been made in this area, and these type systems are increasingly usable and practical. However, their adoption in mainstream programming languages is still limited, which is largely attributed to their strict alias prevention principles that obstruct the usage of existing programming patterns. This is a deterrent to the migration of existing code bases. To tackle this problem, we propose Capture Separation Calculus (System CSC), a calculus that models fork-join parallelism and statically prevents data races while being compatible with established programming patterns. It follows a *control-as-you-need* philosophy: by default, aliases are allowed, but they are tracked in the type system. When data races are a concern, the tracked aliases are controlled to prevent data-race-prone patterns. We study the formal properties of System CSC. Type soundness is proven via the standard progress and preservation theorems. Additionally, we formally verify the data race freedom property of System CSC by proving that the reduction of a well-typed program is confluent.

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

Additional Key Words and Phrases: Scala, capture checking, type systems, safe concurrency, data race freedom

1 INTRODUCTION

Data races are a notorious problem in concurrent programming. They occur when multiple concurrent threads access the same state, with at least one of them mutating it. Programs with data races execute non-deterministically, and data race conditions are often subtle. This makes data-race-related bugs especially challenging to detect and fix.

Over recent decades, there has been a lot of literature on mitigating data races. The heart of data races lies in unsynchronised sharing of mutable state. There has been significant research interest in developing type systems that prevent data races. Many approaches use the mechanism known as *alias control* [Clebsch et al. 2015; Gordon et al. 2012; Reynolds 1978; Servetto et al. 2013]. These systems track and control mutable state aliasing in order to forbid data sharing patterns that are prone to data races. Consider Rust's ownership system [Klabnik and Nichols 2018; Weiss et al. 2019] as an example:

```
let mut v = vec![1, 2, 3];
let handle = thread::spawn(move || { v.push(4); });
println!("Now we have {} elements", v.len());
  // error: borrow of moved value: `v`
```

There is a data race condition in the code: the spawned thread mutates `v` while the main thread reads it. Rust reports a type error on the last line, because its ownership system mandates that a spawned thread should take the full ownership of data it captures, which precludes any other simultaneous aliases.

Authors' addresses: Yichen Xu, EPFL, Lausanne, Switzerland, yichen.xu@epfl.ch; Aleksander Boruch-Gruszecki, EPFL, Lausanne, Switzerland, aleksander.boruch-gruszecki@epfl.ch; Martin Odersky, EPFL, Lausanne, Switzerland, martin.odersky@epfl.ch.

While recent type systems designed to prevent data races have been increasingly usable and practical, their adoption in mainstream programming languages remains limited. This can be attributed to various factors, among which an essential one is that the restrictions imposed by their alias prevention principles obstruct the usage of existing programming patterns[1] [Ferdowsi 2023; Levy et al. 2015]. Consider the following code in Rust:

```
struct Vec2 { x: i32, y: i32 }
fn update<F, G>(p: &mut Vec2, mut f: F, mut g: G)
  where F: FnMut(&i32) -> i32, G: FnMut(&i32) -> i32 {
    p.x = f(&p.x); p.y = g(&p.y);
}
fn main() {
    let mut p = Vec2 { x: 1, y: 2 };
    let mut sum = 0;
    update(&mut p, |&x| { sum += x; x + 1 }, |&y| { sum += y; y + 1 });
}
```

Here, the `Vec2` struct represents a two-dimensional integer vector. The `update` function modifies both dimensions of `p` by calling the two closures `f` and `g`. In the `main` function, we increment both dimensions of `p` by one and compute the sum of the two dimensions simultaneously. Arguably, this is a reasonable programming pattern. In the last line of `main`, although both closures retain a mutable alias to `sum`, the two closures execute sequentially, thus being devoid of data races. Yet, Rust rejects this line because the first mutable borrow to `sum` (as in the first closure) prevents further aliases to `sum` from coexisting.

Since these mechanisms do not align well with established programming patterns, they necessitate a significant paradigm shift and substantial code refactoring when being applied to existing code bases. Why do these systems challenge existing programming patterns? One essential reason is that most of them enforce a *global alias prevention invariant*. Consider Rust: it ensures the uniqueness of owned and mutably borrowed references [Klabnik and Nichols 2018]. Similarly, for balloon types [Servetto et al. 2013] and reference capabilities [Gordon et al. 2012], they mandate that the reference to an isolated object graph is globally unique. Mutable Variable Semantics (MVS) [Racordon et al. 2022] by default uses copy semantics and disallows aliases, with `inout` parameters permitting aliases only in a second-class and scoped manner. These invariants assert that certain data or references are globally unique, i.e., unaliased, in the program. We acknowledge that the benefits of these approaches extend beyond data race freedom: for instance, Rust's ownership and borrowing system is instrumental in providing safe and efficient memory management. However, the flip side is the clash with established programming norms. Given that aliases have long been a mainstay in programming, stringent prevention of them naturally disrupts existing code patterns.

Although existing code bases will benefit greatly from static race freedom, the challenge of aligning with existing programming patterns is a significant deterrent to adopting current alias prevention systems. Given that the functionalities of many existing methods often span beyond mere race freedom–addressing aspects like resource and memory management–it is conceivable that a system with a more narrow focus on data race freedom could be less restrictive and more compatible with existing programming patterns. With this perspective, we aim to develop a type system that:

- statically ensures **data-race-free concurrency**,
- and is **flexible** enough so that it is **compatible** with programming patterns that involve aliases.

[1] A well-known example is the implementation of doubly linked lists in Rust: https://rust-unofficial.github.io/too-many-lists/.

We propose Capture Separation Calculus (System CSC). It is based on Capture Calculus (System $CC_{<:\square}$) [Odersky et al. 2022], a calculus developed recently for effect and resource checking. CSC extends the semantics of $CC_{<:\square}$ with fork-join parallelism, and its type system with mechanisms for static prevention of data races. The central idea of $CC_{<:\square}$ is to track captured variables in types, for which it introduces the concept of *capturing types*. Capturing types have the form $T\,\hat{}\,\{x_1, \cdots, x_n\}$, where T is the shape type of a value (e.g. an integer, or a function), and $\{x_1, \cdots, x_n\}$ is the *capture set* that over-approximates the set of variables captured by the value. For instance, `(() -> Unit)^{file}` describes a function that captures a file handle. For function types, we use `A ->{`$x_1, \cdots, x_n$`} B` as a shorthand for `(A -> B)^{`$x_1, \cdots, x_n$`}`. Moreover, `A => B` is a syntactic sugar for `(A -> B)^{cap}`.

What motivates our choice to base our system on $CC_{<:\square}$? Most importantly, capturing types can be used as a device for *descriptive alias tracking*. Firstly, the capture set essentially describes what a value at most aliases, based on which alias controlling mechanisms can be developed. Secondly, capturing types are descriptive (rather than prescriptive) in the sense that they faithfully track captures in types but barely enforce any restrictions on them. There are a few pieces of evidence suggesting that $CC_{<:\square}$ is practical and ergonomic. $CC_{<:\square}$ has been implemented as part of Scala 3 compiler.[2] Besides, the Scala 3 compiler itself can be successfully capture checked.[3] Furthermore, Part of Scala's standard library has been ported to be capture-checking-compatible.[4] These factors make $CC_{<:\square}$ an ideal basis for CSC. Based on the alias tracking framework of $CC_{<:\square}$, CSC develops alias prevention mechanisms and parallelism constructs to support race-free concurrency.

CSC follows a *control-as-you-need* philosophy for alias prevention: it permits aliases by default, but steps in to regulate them when data races are a concern. For instance, the following is the pseudo-Scala equivalent of the Rust example provided earlier:

```
case class Vec2(var x: Int, var y: Int)
def update(p: Vec2, f: (a: Int) => Int, g: (a: Int) => Int): Unit =
  p.x = f(p.x); p.y = g(p.y)
def main(): Unit =
  val p = new Vec2(1, 2)
  val sum = new Ref(0)
  update(p, x => { sum.update(old => old + 1); x + 1 },
            y => { sum.update(old => old + 1); y + 1 })
```

The `update` method of `sum` updates its content, with the function passed to it computing the new value from the old one. This code, being well-typed, remains so upon the introduction of CSC without the need of any refactoring. Indeed, the two closures both alias `sum` but are executed sequentially. The aliases are thus harmless in terms of data races. However, the following program *should* be rejected as it *does* suffer from data races:

```
def parupdate(p: Vec2, f: (a: Int) => Int, g: (a: Int) => Int): Unit =
  par(p.x = f(p.x), p.y = g(p.y))
def main(): Unit =
  // ...
  parupdate(p, x => { sum.update(_ + 1); x + 1 },
               y => { sum.update(_ + 1); y + 1 })
```

In the code, `par(`t_1`, `t_2`)` runs the two operations in parallel. The only difference between `update` and `parupdate` is that the latter one updates the two fields concurrently. In fact, the expression `par(p.x = f(p.x), p.y = f(p.y))` is ill-typed in CSC, because, to parallelize two operations, the

[2] https://dotty.epfl.ch/docs/reference/experimental/cc.html
[3] https://github.com/lampepfl/dotty/pull/16292
[4] https://github.com/lampepfl/dotty/pull/18192

separation between their captures must be established. To fix this, the `parupdate` function should declare its arguments to be separated:

```
def parupdate(p: Vec2,
      sep{p} f: (a: Int) => Int,
    sep{p,f} g: (a: Int) => Int): Unit = ...
```

Here, **sep**`{p} f: ...` indicates that `f` should be *separated* from `p`. The set `{p}` is called *separation degree*. In CSC, separation does not mean that the two parties reference disjoint states. Instead, it means *non-interference*: if `f` is separated from `p`, there is no mutable variable referenced by both and mutated by either of them. Similarly, **sep**`{p,f}` indicates that `g` should be separated from both `p` and `f`. With the separation annotations, the type error shifts to the call site of `parupdate`: it is required that `g` is separated from `f`, but both closures mutate `sum`. The two examples show the flexibility of CSC: aliases are tracked but permitted by default to remain compatible with existing patterns. Yet, when users opt for safe parallelism constructs, mutable state aliases are controlled.

To validate the practical applicability of CSC, we implement a prototype in the form of an extension to the Scala 3 compiler [Scala 2022]. Moreover, the prototype supports *separation degree inference*. In practice, this eliminates the need for users to explicitly specify the separation degrees for function arguments (colored in gray in the example above):

```
def parupdate(p: Vec2,
         sep f: (a: Int) => Int,
         sep g: (a: Int) => Int): Unit = ...
```

Internally, when typing the body of a function, the compiler gathers constraints on the omitted separation degrees and solves the constraints incrementally. The inference is local: once a function is typed, the inferred separation degrees of its arguments are frozen.

In the metatheory, we establish the type soundness of CSC with the standard progress and preservation theorems. Furthermore, we prove that the reduction of a well-typed CSC program is *confluent*. Essentially, this result implies that CSC prevents data races statically.

2 CONTROL AS YOU NEED

The problem of data race prevention is fundamentally a problem of controlling aliases to mutable state. CSC controls these aliases *as you need*. By default, it allows aliases to mutable state, but keeps track of them through types. Then, when the need for data race prevention arises, it restricts the previously-tracked aliases to reject aliasing patterns that cause data races. In this section, we will develop the core concepts of CSC piece by piece.

2.1 Capturing Types for Descriptive Alias Tracking

We start by introducing capturing types, the alias tracking device in CSC. They are an idea proposed in $CC_{<:\square}$ [Odersky et al. 2022].

Originally proposed for effect tracking, capturing types have the form $T\,\hat{}\,\{x_1, \cdots, x_n\}$, where

- T is the *shape* type, which describes the shape of a value, e.g. a function mapping `String` to `Unit`,
- and $\{x_1, \cdots, x_n\}$ is the *capture set*, which is an upper bound of the variables captured by the value.

$CC_{<:\square}$ takes an object-capability-based perspective towards effect tracking [Miller 2006; Odersky et al. 2022]. In $CC_{<:\square}$, programs perform effects through object capabilities, and object capabilities are represented as regular program variables. Given that the capture sets in capturing types predict captured variables of values, CSC is capable of reasoning about the set of capabilities a closure at most accesses, and thus the effects it at most performs. Take the following closure as an example:

```
val sayHello = (name: String) => console.log("Hello, " + name)
```

Assume that `console` is a capability for console I/O. The closure writes to the console through the `console` capability. It is typed as `String ->{console} Unit`, which signifies that the closure can at most perform console I/O.

To capture is to retain a reference, or an alias, to a variable. Therefore, capturing types track aliases as well: the capture set of a closure is the set of variables it at most captures references to.

```
val sum = new Ref(0)
val incr = () => { sum += 1; sum }
```

In this example, `sum` is a mutable variable and `incr` is a function that increments it. `incr` has the type `(() -> Unit)^{sum}`, which tracks the function's reference to `sum` in the capture set. In fact, one can view accessing a mutable variable as an effect, and the variable itself as a capability for that effect. In that sense, to track mutable state aliases and to track the effects for accessing mutable state are essentially equivalent.

When tracking mutable state aliases for data race prevention, it is useful to differentiate between *immutable* and *mutable* references. It adds to the flexibility of the system: a mutable variable is allowed to be referenced by multiple threads as long as all references are immutable. The *capability hierarchy* in capturing types can be used for that. In $CC_{<:\square}$, any capability is derived from a set of existing capabilities, obtaining the authority to perform effects from them, and thus forming a hierarchy. Sitting at the root of the hierarchy is the universal capability **cap**, from which all capabilities are utimately derived.

```
class Counter(x: Ref[Int]^{cap}):
  def incr(): Unit = x.update(_ + 1)
  def get: Unit = x.get
val x: Ref[Int]^{cap} = new Ref(0)
val counter: Counter^{x} = new Counter(x)
```

In this example, the `Counter` class creates a counter capability from a mutable variable. We derive `counter` from `x`. The derived capability `counter` has no more authority for performing effects other than that obtained from `x`. Notice that in the example above, the mutable variable `x` is derived from the universal capability **cap**. In the actual system of CSC, we introduced the reference root capability **ref**, which is a special universal capability from which all mutable variables are derived. In other words, when allocating a new mutable variable `x` in CSC like the example before, the type of `x` is `Ref[Int]^{ref}`.

The derivation of capabilities aligns with the relation between immutable and mutable references. Through a mutable reference one can either read from or write to a variable, while an immutable one can only perform read. If we view a mutable variable as a *full* capability for mutably accessing itself, then it is natural to view an immutable reference as a capability derived from the full one that can only perform reads. Inspired by this idea, CSC introduces *reader capabilities*, which are capabilities derived from mutable variables, holding only authorities for reading from them. The idea is similar to that of fractional permissions [Boyland 2010].

The most important design goal of CSC is the compatibility with existing programming patterns. Therefore, as a basis of CSC, $CC_{<:\square}$ itself should be flexible and lightweight enough: it should not be too restrictive to express common patterns, and introducing it should only require minimal additional annotations. Thankfully, $CC_{<:\square}$ is ergonomic enough to be integrated into an existing language with reasonable effort. One reason is that capturing types are descriptive: they faithfully track captures in types with minimal restrictions enforced [Odersky et al. 2022]. Furthermore, $CC_{<:\square}$ expresses *effect polymorphism* in a lightweight yet expressive manner. For example, after introducing effect checking, the list map function should be polymorphic over the effects performed

by the mapping function. This is a long-standing problem in effect systems [Brachthäuser et al. 2020; Leijen 2017; Odersky et al. 2022], and often requires refactoring existing function signatures. How does $CC_{<:\square}$ express that?

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

Surprisingly, it looks *exactly* the same as what it was in Scala's standard library. Recall that `A => B` stands for `(A -> B)^{cap}`. This type ranges over functions that perform effects through the **cap** capability. Based on the capture hierarchy, *any* capability ultimately obtains its authority from **cap**, and therefore *any* effect is ultimately performed through **cap**. So `A => B` effectively ranges over functions that perform any effect: the map function is effect polymorphic.

These advantages of $CC_{<:\square}$ are inherited by CSC. Capturing types enable descriptive alias tracking, which by default does not restrict established aliasing patterns. Furthermore, CSC naturally expresses in a lightweight manner higher-order functions that are polymorphic over the aliases their arguments may capture.

2.2 Separation for Flexible Alias Prevention

We have seen how CSC employs capturing types to track aliases. We have seen capturing types in $CC_{<:\square}$ and how it can be employed to track aliases. CSC, which is built on top of $CC_{<:\square}$, inherits its ability for descriptive alias tracking. The next question naturally arises: how does it regulate these tracked aliases to prevent race conditions? To this end, CSC introduces the concept of *separation*, along with two associated formal devices: *separation degrees* and *separation checking*.

A separation degree specifies a set of variables that a variable is *separated* from. Separation in CSC does not mean the disjointness of memory locations used by two parties. Instead, it means *non-interference*: if a mutable variable is referenced by both parties, then both references are read-only. The following two functions construct a pair given two elements that are lazily evaluated. However, `seq` evaluates the two elements sequentially, while `par` evaluates them in parallel.

```
def seq[A, B](op1: () => A, op2: () => B): (A, B) =
  (op1(), op2())
def par[A, B](op1: () => A, sep{op1} op2: () => B): (A, B) =
  letpar a = op1() in
    (a, op2())
```

The part highlighted in grey denotes a separation degree. Here, it signifies that the second argument `op2` should be *separated* from the first argument, `op1`. This ensures that the two operations `op1` and `op2` can be evaluated in parallel safely in `par`.

The **letpar** $x = t$ **in** u form is a parallel let expression. Its binding and body terms are evaluated in parallel. It has a form of fork-join parallelism, and its semantics is similar to futures: viewing x as a future that runs t, mentioning x in u is like awaiting the future x, which blocks the evaluation of u until x is evaluated to a value.

When a parallel let expression is type-checked, separation checking ensures the separation between the binding and the continuation. The separation check here is the essence of CSC's data race prevention mechanism: the terms that are evaluated in parallel at runtime are statically guaranteed to be non-interfering with each other.

When type-checking a function application, separation checking ensures the separation degree declared by the formal parameter is respected. For instance, we can use `par` to parallelize two updates to disjoint mutable variables:

```
val a = new Ref(0)
val b = new Ref(0)
par(   () => a.set(42),    () => b.set(0)   )
```

captures $\{a\}$ should be separated from $\{a\}$

When type-checking the second argument, separation checking ensures that the declared separation degree, `{op1}`, is respected, which, in this case, requires the separation between `{a}` and `{b}`.

It is straightforward to see that `{a}` can be safely assumed to be separated from `{b}`: both are freshly allocated mutable variables and therefore not aliased by anything else. But how is this *freshness* expressed in the formalism? In CSC, one can arbitrarily specify the separation degree of a freshly-allocated mutable variable, possibly declaring it to be separated from anything else in the context. This mirrors the fact that a fresh mutable variable is not aliased by anything else.

```
val a = new Ref(0)
val b = new Ref(true)
val c = new Ref("Hello")
```

For instance, in the code above, one can declare the separation degree of `a` to be empty, that of `b` to be `{a}`, and that of `c` to be `{a, b}`. Note that, although these declared separation degrees are assymetric, separation checking is symmetric and the separation between any two of `a`, `b` and `c` can be established.

Separation checking, as suggested by its name, checks whether two sets of variables are separated. It is a symmetric relation over two capture sets. Separation between two capture sets $\{x_1, \cdots, x_n\}$ and $\{y_1, \cdots, y_m\}$ is established by the pairwise separation between variables in the two sets, i.e. x_i being separated from y_j for all $i \in \{1, \cdots, n\}$ and $j \in \{1, \cdots, m\}$. To establish the separation between two variables, separation checking makes use of separation degrees, and follows the aliases tracked by capturing types. Specifically, if x has the separation degree $\{y_1, \cdots, y_n\}$, then any y_i for $i \in \{1, \cdots, n\}$ is considered separated from x, and vice versa. For instance, in the previous code snippet, the separation between any two among `a`, `b` and `c` can be established using the declared separation degrees.

The following example, which is a continuation of the previous code snippet, illustrates how separation checking follows aliases:

```
val counter : Counter^{a}          = new Counter(a)
val f       : () ->{counter} Unit  = () => counter.incr()
val g       : () ->{b,c}     Unit  = () => {
  b.set(false)
  c.update(s => s + " World")
}
par(f, g)  // ok
```

The text colored in grey shows the type of each variable in the context. For the last line to type-check, the separation between `f` and `g` should be established. It works as follows:

- Neither `f` nor `g` has a separation degree attached, so we follow their aliases tracked in the types. Hence, we intend to establish the separation between `{counter}` and `{b, c}`.
- That requires `counter` to be separated from both `b` and `c`. We take the separation between `counter` and `b` as an example. The other case is similar.
- We follow the alias of `counter` again, which is `a`. So the goal becomes the separation between `a` and `b`.
- It holds, as `a` is in `b`'s separation degree.

Separation is the formal device for data race freedom in CSC: given two programs t_1 and t_2 and their captured variables $\{x_1, \cdots, x_n\}$ and $\{y_1, \cdots, y_m\}$, executing t_1 and t_2 in parallel will not cause data races if the separation between $\{x_1, \cdots, x_n\}$ and $\{y_1, \cdots, y_m\}$ is established. Separation degrees track separations in the context, whereas separation checking enforces separations between terms.

2.3 A Case Study

Now we study an example to see how CSC works in action.

```
class Loss(alpha: Double):
  val invokes: Ref[(Int, Int)]^{ref} = new Ref((0, 0))
  val recordL1: () ->{invokes} Unit = () =>
    invokes.update((n1, n2) => (n1 + 1, n2))
  val recordL2: () ->{invokes} Unit = () =>
    invokes.update((n1, n2) => (n1, n2 + 1))

  def l1Loss(output: Tensor, truth: Tensor): Tensor =
    val result =
      // ... compute l1 loss ...
    recordL1()
    result

  def l2Loss(output: Tensor, truth: Tensor): Tensor =
    val result =
      // ... compute l2 loss ...
    recordL2()
    result

  def combinedLoss(output: Tensor, truth: Tensor): Tensor =
    val l1 = l1loss(output, truth)
    val l2 = l2loss(output, truth)
    alpha * l1 + (1.0 - alpha) * l2
```

Listing 1. A class that computes loss functions. It computes L1 loss, L2 loss and a weighted average of them. Also, it profiles how many times L1 and L2 losses are computed.

Listing 1 shows a class that compute loss functions in a hypothetical deep learning framework [Goodfellow et al. 2016]. It assumes a standard tensor implementation `Tensor` which supports regular tensor operations. Its methods `l1Loss` and `l2Loss` compute the L1 and L2 loss functions, respectively [Goodfellow et al. 2016]. The `combinedLoss` method, on the other hand, computes the weighted average of the two losses. The class also profiles how many times L1 and L2 losses are computed. To do that, it has a mutable variable `invokes` which counts the invocations of the two loss functions. `recordL1` and `recordL2` increment the counter for the two losses respectively.

By far, the code works as expected and is devoid of data races. It is well-typed in CSC. The method `l1Loss` has the type `(Tensor, Tensor) ->{recordL1} Tensor`, and `l2Loss` has the type `(Tensor, Tensor) ->{recordL2} Tensor`. Although both `l1Loss` and `l2Loss` capture mutable aliases to the `invokes` variable, they are used sequentially in `combinedLoss` and do not cause a race condition.

What if one wants to accelerate the computation of `combinedLoss` by parallelizing the calculation of the L1 and L2 loss? Listing 2 shows a parallelized version of `combinedLoss`. Although it is subtle,

this method is indeed subject to data races. It is because `l1Loss` and `l2Loss` are parallelized, but both of them mutate the mutable variable `invokes`.

```
class Loss(alpha: Double):
  /* ... same as above ... */
  def combinedLossFast(output: Tensor, truth: Tensor): Tensor =
    letpar l1 = l1Loss(output, truth) in
      val l2 = l2Loss(output, truth)
      l1 * alpha + l2 * (1 - alpha)
```

Listing 2. A new method in `Loss` which parallelize loss calculations for acceleration.

This race-prone state sharing pattern will be detected and reported by CSC, as the separation check for the **`letpar`** form will fail:

- To succeed the separation check, the separation between `l1Loss` and `l2Loss` needs to be established.
- By following aliases, this ends up requiring the separation between `invokes` and itself, which is false.

```
class Loss(alpha: Double)
  val invokesL1: Ref[Int]^{ref} = new Ref(0)
  val invokesL2: Ref[Int]^{ref} = new Ref(0)
  val recordL1: () ->{invokesL1} Unit =
    () => invokesL1.update(_ + 1)
  val recordL2: () ->{invokesL2} Unit =
    () => invokesL2.update(_ + 1)

  /* ... same as above ... */
```

Listing 3. One possible refactorization of the code so that it passes separation checks.

The failed separation check provides useful hints for refactoring. Listing 3 shows a possible fix for the code. It splits the mutable variable into two, counting the invocations of the two loss functions separately. Now the code is race-free and well-typed: by following the aliases, the separation between `l1Loss` and `l2Loss` is established from the separation between `invokesL1` and `invokesL2`.

2.4 Reader Capabilities

Now, we have seen how CSC statically prevents data races by regulating aliases to mutable state: it tracks aliases in capturing types, and controls aliases by establishing separations. Modeling immutable aliases adds to the expressiveness of the system: unlike mutable aliases, immutable aliases can safely co-exist in parallel without inducing race conditions. CSC distinguishes between immutable and mutable aliases by introducing *reader capabilities*.

Recall that a mutable variable can be viewed as a capability for mutably accessing itself. In CSC, we allow deriving a reader capability from a mutable variable, which only possesses the authority to read from the mutable variable.

```
val a = new Ref(0)
val r : Rdr[Int]^{a} = a.reader
par(() => a.set(0), () => a.set(42))  // error
```

```
par(() => a.set(0), () => r.get)      // error
par(() => r.get,    () => r.get)      // ok
```

The example shows how reader capabilities work. We first create a mutable variable `a`. Then, we derive a reader capability `r` from it. The type of `r` is `Rdr[Int] ^ {a}`, where

- the shape type `Rdr[Int]` signifies that `r` is a reader capability for a mutable variable that holds an integer;
- and the capture set `{a}` signifies that `r` is derived from `a`.

Separation checking treats reader capabilities differently: since only reads can be performed through a reader capability, and concurrent reads never interfere with each other, separation between reader capabilities can always be established. Recall that the function `par` requires the separation between its two arguments. The first two invocations of `par` are both subject to data races and ill-typed. The first call is ill-typed because `a` is not separated from itself. Typing the second one needs to establish the separation between `a` and `r`, which, following the aliases, requires the separation between `a` and itself, which is false. Conversely, the last invocation of `par` is data-race-free. Though both arguments access the mutable variable `a`, both accesses are through a reader capability and thus read-only. Indeed, the last line is well-typed in CSC: the separation between `r` and itself can be established since `r` is a reader capability.

We have seen that **cap** can be used as a device for alias polymorphism. However, a function of type `A ->{cap} B` could retain aliases to any mutable variable, so it cannot be parallelized with itself. For instance, one may want to implement a `parmap` function which maps a function in parallel on a list:

```
class List[+A]:
  def parmap[B](f: A ->{cap} B): List[B] = this match
    case Nil => Nil
    case x :: xs =>
      letpar x1 = f(x) in
        val xs1 = xs.parmap(f)
        x1 :: xs1
```

It is subject to race conditions. The **letpar** form parallelizes two invocations to `f`. A data race occurs if `f` mutates any mutable variable. The code is ill-typed in CSC: the separation between `f` and itself cannot be established.

However, it is safe to parallelize a function with itself if that function only performs reads. Therefore, it is useful to express polymorphism over read-only functions. To do that, CSC introduces the *reader root* **rdr**.

```
class List[+A]:
  def parmap[B](f: A ->{ rdr } B): List[B] = ...
```

After changing the signature of `parmap` to the above, the code complies, and data race freedom is still statically guaranteed. A function of type `A ->{rdr} B` could reference any reader capability. In other words, it can *read* arbitrary mutable state. Parallelizing such a function with itself is indeed data-race-free.

3 A CALCULUS FOR SAFE CONCURRENCY

In this section, we present the formal definitions of CSC. It is based on $CC_{<:\Box}$, which itself is based on System $F_{<:}$ in monadic normal form, where most syntactical structures are standard. Figure 1 and 2 show the syntax and typing rules of CSC, with the changes from $CC_{<:\Box}$ highlighted in gray.

Type Variable	X, Y, Z		
Variable	x, y, z, **cap**, **rdr**, **ref**		
Let Mode	m	::=	ϵ | par
Value	v, w	::=	$\lambda(x :_D T)t$ | $\lambda[X <: S]t$ | $\square\, x$ | reader x
Answer	a	::=	v | x
Term	s, t	::=	a | $x\, y$ | $x\,[S]$ | $\mathsf{let}_m\, x = s\, \mathsf{in}\, t$ | $C \multimap x$ | $\mathsf{var}_D\, x := y\, \mathsf{in}\, t$ | $\mathsf{read}\, x$ | $x := y$
Shape Type	S	::=	X | $\top$ | $\forall(x :_D U)T$ | $\forall[X <: S]T$ | $\square\, T$ | $\mathsf{Ref}[S]$ | $\mathsf{Rdr}[S]$
Type	T, U	::=	S | $S \wedge C$
Capture Set	C	::=	$\{x_1, \cdots, x_n\}$
Separation Degree	D	::=	C if **cap**, **ref**, **rdr** $\notin C$
Typing Context	Γ	::=	$\emptyset$ | $\Gamma, X <: S$ | $\Gamma, x :_D T$
Store	γ	::=	$\cdot$ | $\gamma, \mathsf{val}\, x = v$ | $\gamma, \mathsf{var}\, x := v$ | $\gamma, \mathsf{set}\, x := v$
Evaluation Context	e	::=	$[]$ | $\mathsf{let}_m\, x = e\, \mathsf{in}\, u$ | $\mathsf{let}_{\mathsf{par}}\, x = t\, \mathsf{in}\, e$

Fig. 1. Syntax of CSC.

3.1 Syntax

Monadic normal form. CSC inherits the monadic normal form (MNF) from $\mathrm{CC}_{<:\square}$, where operands in a application are restricted to variables [Odersky et al. 2022]. This does not lead to loss of expressiveness: a general application $t_1\, t_2$ can be encoded in MNF as $\mathsf{let}\, x = t_1\, \mathsf{in}\, \mathsf{let}\, y = t_2\, \mathsf{in}\, x\, y$. The monadic normal form simplifies the formalism. Given that capture sets and separation degrees could mention term variables, the types in CSC are reference-dependent. Thanks to MNF, typing applications only involve variable renaming in the result types. MNF is also adopted in Dependent Object Types (DOT), the theoretical foundation of Scala, for similar reasons.[Amin et al. 2016; Rapoport and Lhoták 2019]

Capturing types. The syntactic forms of types in CSC are mostly the same as in $\mathrm{CC}_{<:\square}$, stratified into shape types S and capturing types $S \wedge C$. The capture set C is a set of program variables including the three root capabilities **cap**, **ref** and **rdr**. The shape types inherit the type constructors in System $\mathrm{F}_{<:}$: type variables, the top type, funtion types, and polymorphic types. Shape types that are introduced by $\mathrm{CC}_{<:\square}$ and CSC will be explained later. We freely use a shape type S where a capturing type is expected with the isomorphism $S \equiv S \wedge \{\}$.

Mutable variables. $\mathsf{var}_D\, x := y\, \mathsf{in}\, t$ allocates a new mutable cell. It initializes the cell with the value of y, and binds the reference to the cell as x in t. The shape type $\mathsf{Ref}[S]$ describes a reference to a mutable cell whose content is of type S. It declares a separation degree D for the mutable variable. It can be chosen arbitrarily, possibly spanning over the entire domain of the environment. This reflects the *freshness* of the newly-allocated variable: it is separated from everything else in the program. The form reader x derives a reader capability from the mutable variable x, and $\mathsf{Rdr}[S]$ is the shape type for readers. The forms $x := y$ and $\mathsf{read}\, x$ write to and read from mutable variables respectively.

Separation degrees. Like a capture set, a separation degree is a set of program variables, but it syntactically excludes the three root capabilities **cap**, **ref** and **rdr**. This is because a separation degree with root capabilities in it does not make sense. All term bindings in CSC carry a separation

Subcapturing $\Gamma \vdash C_1 <: C_2$

$$\frac{\Gamma \vdash C_1 <: C_2 \quad \Gamma \vdash C_2 <: C_3}{\Gamma \vdash C_1 <: C_3}\ (\textsc{sc-trans}) \qquad \frac{x : S\,\hat{}\,C \in \Gamma}{\Gamma \vdash \{x\} <: C}\ (\textsc{sc-var}) \qquad \frac{\Gamma \vdash \{x_1\} <: C_2 \quad \cdots \quad \Gamma \vdash \{x_n\} <: C_2}{\Gamma \vdash \{x_1, \cdots, x_n\} <: C_2}\ (\textsc{sc-set})$$

$$\frac{x \in C}{\Gamma \vdash \{x\} <: C}\ (\textsc{sc-elem}) \qquad \Gamma \vdash \{\mathbf{rdr}\} <: \{\mathbf{cap}\}\ (\textsc{sc-rdr-cap}) \qquad \frac{\text{is-reader}_\Gamma\ x}{\Gamma \vdash \{x\} <: \{\mathbf{rdr}\}}\ (\textsc{sc-reader})$$

$$\Gamma \vdash \{\mathbf{ref}\} <: \{\mathbf{cap}\}\ (\textsc{sc-ref-cap})$$

where $\text{is-reader}_\Gamma\ x \triangleq (x : T \in \Gamma) \wedge (\Gamma \vdash T <: \mathrm{Rdr}[S]\,\hat{}\,C)$ for some T, C, S

Subtyping $\Gamma \vdash T <: U$

$$\Gamma \vdash T <: T\ (\textsc{refl}) \qquad \Gamma \vdash T <: \top\ (\textsc{top}) \qquad \frac{\Gamma \vdash C_1 <: C_2 \quad \Gamma \vdash S_1 <: S_2}{\Gamma \vdash S_1\,\hat{}\,C_1 <: S_2\,\hat{}\,C_2}\ (\textsc{capt})$$

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3}\ (\textsc{trans}) \qquad \frac{\Gamma \vdash U_2 <: U_1 \quad \Gamma, x :_D U_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x :_D U_1)T_1 <: \forall(x :_D U_2)T_2}\ (\textsc{fun})$$

$$\frac{X <: S \in \Gamma}{\Gamma \vdash X <: S}\ (\textsc{tvar}) \qquad \frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, X <: S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall[X <: S_1]T_1 <: \forall[X <: S_2]T_2}\ (\textsc{tfun}) \qquad \frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash \Box T_1 <: \Box T_2}\ (\textsc{boxed})$$

Separation Checking $\Gamma \vdash C_1 \bowtie C_2$

$$\frac{\Gamma \vdash C_1 \bowtie C_2}{\Gamma \vdash C_2 \bowtie C_1}\ (\textsc{ni-symm}) \qquad \frac{\Gamma \vdash \{x_1\} \bowtie C_2 \quad \cdots \quad \Gamma \vdash \{x_n\} \bowtie C_2}{\Gamma \vdash \{x_1, \cdots, x_n\} \bowtie C_2}\ (\textsc{ni-set})$$

$$\frac{x :_D T \in \Gamma \quad y \in D}{\Gamma \vdash \{x\} \bowtie \{y\}}\ (\textsc{ni-degree}) \qquad \frac{x :_D S\,\hat{}\,C \in \Gamma \quad \Gamma \vdash C \bowtie \{y\}}{\Gamma \vdash \{x\} \bowtie \{y\}}\ (\textsc{ni-var}) \qquad \frac{\Gamma \vdash \{x\} <: \{\mathbf{rdr}\} \quad \Gamma \vdash \{y\} <: \{\mathbf{rdr}\}}{\Gamma \vdash \{x\} \bowtie \{y\}}\ (\textsc{ni-reader})$$

Typing $\Gamma \vdash t : T$

$$\frac{x : S\,\hat{}\,C \in \Gamma}{\Gamma \vdash x : S\,\hat{}\,\{x\}}\ (\textsc{var}) \qquad \frac{\Gamma, X <: S \vdash t : T \qquad \Gamma \vdash S\ \mathbf{wf}}{\Gamma \vdash \lambda[X <: S].t : \forall[X <: S]T\,\hat{}\,\mathrm{cv}\,(t)}\ (\textsc{tabs}) \qquad \frac{\Gamma \vdash x : S\,\hat{}\,C \quad C \subseteq \mathrm{dom}(\Gamma) \cup \{\mathbf{ref}\}}{\Gamma \vdash \Box x : \Box S\,\hat{}\,C}\ (\textsc{box})$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T <: U}{\Gamma \vdash t : U}\ (\textsc{sub}) \qquad \frac{\Gamma \vdash x : \forall(z :_D T)U\,\hat{}\,C \quad \Gamma \vdash y : T \quad \Gamma \vdash \{y\} \bowtie D}{\Gamma \vdash x\ y : [z \mapsto y]\,U}\ (\textsc{app}) \qquad \frac{\Gamma \vdash x : \Box S\,\hat{}\,C \quad C \subseteq \mathrm{dom}(\Gamma) \cup \{\mathbf{ref}\}}{\Gamma \vdash C \multimap x : S\,\hat{}\,C}\ (\textsc{unbox})$$

$$\frac{\Gamma, x :_D U \vdash t : T \quad \Gamma \vdash U\ \mathbf{wf} \qquad \Gamma \vdash D\ \mathbf{wf}}{\Gamma \vdash \lambda(x :_D U).t : \forall(x :_D U)T\,\hat{}\,\mathrm{cv}\,(t) \backslash x}\ (\textsc{abs}) \qquad \frac{\Gamma \vdash x : \forall[X <: S]T\,\hat{}\,C}{\Gamma \vdash x[S] : [X \mapsto S]\,U}\ (\textsc{tapp}) \qquad \frac{\Gamma \vdash s : T \quad \Gamma, x :_{\{\}} T \vdash t : U \qquad x \notin \mathrm{fv}(U) \quad \Gamma \vdash s \bowtie t \quad \text{if } m = \mathrm{par}}{\Gamma \vdash \mathsf{let}_m\ x = s \text{ in } t : U}\ (\textsc{let})$$

$$\frac{\Gamma \vdash y : S \quad \Gamma, x :_D \mathrm{Ref}[S]\,\hat{}\,\{\mathbf{ref}\} \vdash t : U \quad x \notin \mathrm{fv}(U) \qquad \Gamma \vdash D\ \mathbf{wf}}{\Gamma \vdash \mathsf{var}_D\ x := y \text{ in } t : U}\ (\textsc{dvar}) \qquad \frac{\Gamma \vdash x : \mathrm{Ref}[S]\,\hat{}\,C}{\Gamma \vdash \mathrm{reader}\ x : \mathrm{Rdr}[S]\,\hat{}\,\{x\}}\ (\textsc{reader})$$

$$\frac{\Gamma \vdash x : \mathrm{Rdr}[S]\,\hat{}\,C}{\Gamma \vdash \mathrm{read}\ x : S}\ (\textsc{read}) \qquad \frac{\Gamma \vdash x : \mathrm{Ref}[S]\,\hat{}\,C \quad \Gamma \vdash y : S}{\Gamma \vdash x := y : S}\ (\textsc{write})$$

Reduction $\gamma \mid t \longmapsto \gamma' \mid t'$

$$\frac{\gamma(\mathrm{val}\ x) = \lambda(z :_D T)t}{\gamma \mid e\,[\,x\ y\,] \longmapsto \gamma \mid e\,[\,[y/z]t\,]}\ (\textsc{apply}) \qquad \frac{\gamma(\mathrm{val}\ x) = \Box\, y}{\gamma \mid e\,[\,C \multimap x\,] \longmapsto \gamma \mid e\,[\,y\,]}\ (\textsc{open})$$

$$\frac{\gamma(\mathrm{val}\ x) = \lambda[X <: S']t}{\gamma \mid e\,[\,x[S]\,] \longmapsto \gamma \mid e\,[\,[S/X]t\,]}\ (\textsc{tapply}) \qquad \frac{\gamma(\mathrm{val}\ x) = \mathrm{reader}\ y \quad \gamma(\mathrm{var}\ y) = v}{\gamma \mid e\,[\,\mathrm{read}\ x\,] \longmapsto \gamma \mid e\,[\,v\,]}\ (\textsc{get})$$

$$\frac{\mathrm{fv}(v) \subseteq \mathrm{dom}(\gamma)}{\gamma \mid e\,[\,\mathsf{let}_m\ x = v \text{ in } t\,] \longmapsto \gamma, \mathrm{val}\ x = v \mid e\,[\,t\,]}\ (\textsc{lift-let}) \qquad \gamma \mid e\,[\,\mathsf{let}_m\ x = y \text{ in } t\,] \longmapsto \gamma \mid e\,[\,[y/x]t\,]\ (\textsc{rename})$$

$$\frac{\gamma(\mathrm{val}\ y) = v}{\gamma \mid e\,[\,\mathsf{var}_D\ x := y \text{ in } t\,] \longmapsto \gamma, \mathrm{var}\ x := v \mid e\,[\,t\,]}\ (\textsc{lift-var}) \qquad \frac{\gamma(\mathrm{val}\ y) = v}{\gamma \mid e\,[\,x := y\,] \longmapsto \gamma, \mathrm{set}\ x := v \mid e\,[\,v\,]}\ (\textsc{lift-set})$$

Fig. 2. Typing and Evaluation Rules of System CSC

degree. For instance, the function value $\lambda(x :_D T)t$, the function type $\forall(x :_D T)U$, and the term bindings in typing contexts $\Gamma, x :_D T$.

Boxes. Boxes are a construct introduced in $CC_{<:\Box}$, type variables are bounded by and instantiated to only shape types (as seen in the forms $\forall[X <: S]T$, $\lambda[X <: S]t$ and $x[S]$). It is shown that this restriction not only simplifies the formalism, but also adds to the expressiveness of capture polymorphism [Odersky et al. 2022]. To recover type polymorphism over capturing types, boxes are introduced: a boxed type $\Box\, T$, which is a shape type, encapsulates a capturing type T. A type variable can therefore be instantiated to a boxed capturing type. There are two box-related expressions: the box form $\Box\, x$ and the unbox form $C \multimap x$ introduce and eliminate a box respectively.

Parallel let bindings. The let form $\text{let}_m\ x = t\ \text{in}\ u$ has a mode m, which is either par or ϵ. par signifies a parallel let binding, where the binding t and the continuation u are evaluated in parallel. It is a form of fork-join parallelism and has a future-like semantics: x is as if bound to a future running t in u, and to mention x is to await that future. When $m = \epsilon$, the expression is a regular sequential let binding.

3.2 Subcapturing and Subtyping

Throughout the presentation and the metatheory of the calculus, we adopt the Barendregt convention where all bound variables are distinct. For instance, given any typing context $\Gamma, x : T$, we assume that $x \notin \text{dom}(\Gamma)$.

The subcapturing rules establish a pre-order between capture sets. Intuitively, the subcapturing relation compares *how much* a capture set captures: the left-hand-side captures *less* than the right-hand-side. In this sense, subset should imply subcapturing, which is established by the rules (SC-ELEM) and (SC-SET). (SC-TRANS) establishes the transitivity of subcapturing. (SC-VAR) reflects the capability hierarchy in $CC_{<:\Box}$ (and CSC). The rule states that a capability $x : S \wedge C \in \Gamma$ is a subcapture of the set of capabilities from which it is derived (i.e. C). For instance, in a context Γ, given a mutable state $x : \text{Ref}[S] \wedge \{\textbf{cap}\} \in \Gamma$, and its reader capability $y : \text{Rdr}[S] \wedge \{x\} \in \Gamma$, we can derive $\Gamma \vdash \{y\} <: \{x\}$. Although there is no such rule for the subcapturing relation $C <: \{\textbf{cap}\}$ for an arbitrary capture set C, it follows directly from the subcapturing hierarchy: all capabilities are ultimately derived from **cap**. Therefore, by repeated application of (SC-SET), (SC-VAR) and (SC-TRANS), we can derive $C <: \{\textbf{cap}\}$ for any capture set C.

The last three rules (SC-REF-CAP), (SC-RDR-CAP) and (SC-READER) are the new rules introduced by CSC. The first rule encodes the subcapturing relation between the reference root capability **ref** and the universal root capability **cap**. The other two deal with the subcapturing relations regarding the reader root capability **rdr**. The rule (SC-RDR-CAP) encodes the subcapturing relation between **rdr** and the universal root capability **cap**. **cap** is the top in the capability hierarchy, being a supercapture of all capture sets. The rule (SC-READER) establishes the subcapturing relation between reader capabilities and the reader root capability **rdr**. It detects whether a variable x is a reader capability by checking whether the type of x is a subtype of the reader trait $\text{Rdr}[S]$.

CSC does not introduce any change to the subtyping rules of $CC_{<:\Box}$. We show the subtyping rules for the completeness of our presentation. The subtyping rules of $CC_{<:\Box}$ extend the subtyping rules in System $F_{<:}$. Compared to System $F_{<:}$, System $CC_{<:\Box}$ adds the rules for comparing subcapturing types and boxed types. The (CAPT) rule states that $S_1 \wedge C_1$ is a subtype of $S_2 \wedge C_2$ if S_1 is a subtype of S_2 and C_1 subcaptures C_2. The (BOXED) rule propagates the subtyping between two types to their boxed versions.

3.3 Separation Checking

The separation checking judgement is a key element of our approach to statically ensuring data race freedom. It is symmetric (the (NI-SYMM) rule). The (NI-SET) together with the (NI-SYMM) rule imply that and the separation between two capture sets C_1 and C_2 is established from the pairwise separation between each element in the two sets, i.e. $\{x_i\} \bowtie \{y_j\}$ for any $x_i \in C_1$ and $y_j \in C_2$. The (NI-DEGREE) rule make use of separation degrees. The (NI-VAR) rule follows aliases: x is separated from y as long as what it captures, or aliases, is separated from y. The last rule, (NI-READER), establishes the separation between two reader capabilities. It uses subcapturing to check whether a variable is a reader capability. $\Gamma \vdash s \bowtie t$ denotes the separation between two terms:

$$\Gamma \vdash s \bowtie t \quad \triangleq \quad \Gamma \vdash \mathrm{cv}(s) \cap \mathrm{dom}(\Gamma) \bowtie \mathrm{cv}(t) \cap \mathrm{dom}(\Gamma)$$

3.4 Typing

The (VAR) rule looks up variable x in the environment Γ. Given the binding $x : S \wedge C$, it types x as $S \wedge \{x\}$. The capture set $\{x\}$ is smaller than C based on the (SC-VAR) rule. The environment type $S \wedge C$ can be recovered with subtyping.

The (ABS) and (TABS) rule types term and type lambdas. $\Gamma \vdash T$ **wf** denotes the well-formedness judgement. It is standard: capture sets in types should only mention variables defined in the environment, along with the root capabilities **cap**, **ref** and **rdr**.

$$\frac{\Gamma \vdash S \ \mathbf{wf} \qquad C \subseteq \mathrm{dom}(\Gamma) \cup \{\mathbf{cap}, \mathbf{ref}, \mathbf{rdr}\}}{\Gamma \vdash S \wedge C \ \mathbf{wf}} \quad \text{(WF-CAPT)}$$

The cv $(\cdot)$ function computes the set of variables captured by a term. And it is used to determine the capture sets in the function types.

Definition 3.1 (Captured Variables). *The definition of cv $(\cdot)$ is given as follows, with the changes from $CC_{<:\Box}$ highlighted in gray:*

$$\begin{array}{lcll}
cv(\lambda(x\colon T)t) & = & cv(t) \setminus x, & \\
cv(\lambda[X <\colon S]t) & = & cv(t), & \\
cv(x) & = & \{x\}, & \\
cv(let_m\ x = v\ in\ u) & = & cv(u), & if\ x \notin cv(u) \\
cv(let_m\ x = t\ in\ u) & = & cv(t) \cup cv(u) \setminus x, & \\
cv(x\ y) & = & \{x, y\}, & \\
cv(x[S]) & = & \{x\}, & \\
cv(\Box\ x) & = & \{\}, & \\
cv(C \multimap x) & = & C \cup \{x\}, & \\
cv(var_D\ x := y\ in\ u) & = & \{y\} \cup cv(u) \setminus x, & \\
cv(x := y) & = & \{x, y\}, & \\
cv(reader\ x) & = & \{x\}, & \\
cv(read\ x) & = & \{x\}. &
\end{array}$$

```
def withFile[T](path: String)(op: (f: File^{cap}) => T): T =
  val f = new File(path)
  val result = op(f)
  f.close()
  result
val content = withFile("path/to/file"): f =>  // ok
  f.read()
val leaked  = withFile("path/to/file"): f =>  // error
  () => f.read()
leaked()
```

Listing 4. Definition of `withFile` in Scala. It creates a local file capability `f` and passes it to an function to operate on it, after which the file is closed.

Compared to the cv $(\cdot)$ function in $CC_{<:\square}$ [Odersky et al. 2022], CSC adds cases for the new syntactic forms and keeps other cases the same. The new cases are highlighted in grey. cv $(\cdot)$ is similar to free variables, except that the box form hides captures, while the unbox form pops the hidden captures out. For $\mathsf{let}_m\ x = v\ \mathsf{in}\ u$, the captures of v is only counted if x is mentioned in u. This rule results in smaller captured variable sets when interacting with boxes. For instance, thanks to this rule, the following term that creates a closure that retains an capability called `io` and boxes the closure has an empty captured variable set:

$$\begin{aligned}&\mathsf{let}_\epsilon\ x_f = \\ &\quad \lambda(z : \top).\mathtt{io} \\ &\mathsf{in}\ \square\ x_f\end{aligned}$$

The (APP) rule types function applications. Importantly, we check whether the declared separation degree D is respected by checking the separation between $\{y\}$ and D. The (TAPP) rule types the type function.

The (BOX) and (UNBOX) rules introduce and eliminate boxed types. They requires that the encapsulated capture set only contains variables bound in the environment or the reference root capability **ref**. Therefore, a boxed value whose capture set under the box includes the root capability cannot be unboxed and used. This is how $CC_{<:\square}$ ensures the scoping of local capabilities [Odersky et al. 2022]. Listing 4 shows an example of a scoped capability. The function `withFile` creates a local file capability `f` for a given path, and passes it to a function to operate on it. Since the file is closed after the operation, it is *local*, and its scope should be confined to the function passed to `withFile`. $CC_{<:\square}$ is able to enforce this scoping by the box/unbox mechanism: it accepts the first usage of `withFile`, but rejects the second one as it leaks the local capability. Allowing values that capture **ref** to be boxed and unboxed implies that mutable references are not scoped capabilities. Indeed, in CSC, the lifetime of a mutable variable is not confined to the scope of the var-definition that introduces it. Mutable variable semantics will be discussed in more details in the next section (Section 3.5).

The (LET) rule types let bindings. It first types the binding term, extends the environment with a new type assumption, then types the continuation. The local variable is always introduced with an empty separation degree, but there is no loss of precision: its separation with other capture sets can be established by following the aliases in the capture set. Note that, to ensure that the result type is well-formed, it cannot mention the locally-bound variable. This rule types both parallel and sequential let bindings. In the parallel case, it checks the separation between the binding term and the body term.

The (DVAR) rule types the variable bindings. Note that we restrict the content of mutable variables to have a pure type (or, shape type). We do so to avoid violating the scoping of local capabilities. If the mutable cell were to contain an impure value, one could leak a local capability by assigning it to the cell, bypassing the scope restrictions imposed by boxes:

```
val x = new Ref(new File("dummy"))
withFile("path/to/file"): f =>
  x.set(f)
```

The separation degree for the new variable can be chosen arbitrarily, reflecting the freshness of the variable. Similar to (LET), the result type cannot mention the locally-bound variable.

The (READER) rule types reader creations. Given a mutable variable x, the type of the created capability captures x, reflecting the fact that the reader is derived from x. The (READ) and (WRITE) rules type reads and writes to mutable variables. The read operation needs a reader capability, while the write operation needs a mutable variable, which can be viewed as a full read-write capability to a mutable cell.

3.5 Reduction

The reduction judgement $\gamma \mid t \longmapsto \gamma' \mid t'$ reduces an evaluation configuration to another. A evaluation configuration consists of a store context and the term being reduced. The store context maps immutable variables to their values, and keeps the initial value and each update to the mutable variables. The semantics for mutable variables will be explained later.

The redex is embedded in an evaluation context. Normally, the evaluation context always focuses on the binding term of a let expression. This is the same as the semantics of lets in $\mathrm{CC}_{<:\Box}$. However, for a parallel let, its continuation can be focused as well. This makes possible interleaved evaluation of binding and continuation terms of parallel lets. This is where the parallelism and nondeterminism in CSC comes from.

Now we inspect how reduction rules act on redexes. The (APPLY) and (TAPPLY) rules are standard. They look up the function in the store and reduce to the function body with the formal parameter substituted with the actual argument. The rule for unboxing (OPEN) is the same as in $\mathrm{CC}_{<:\Box}$. It looks up the value of an immutable variable in the store and then drops the box in it. The (LIFT-LET) acts on a let expression whose binding term has been reduced to a value. Since substituting the bound variable with the reduced value breaks the monadic normal form of terms, the reduced binding is lifted to the store. The (RENAME) rule is applicable on the let bindings whose binding term is reduced to a variable. In this case, instead of lifting the binding to the store, we rename the variables in the body term.

Now we illustrate a concrete example for the reduction of parallel let bindings. Let the following be the initial evaluation configuration:

$$\begin{aligned}\gamma \mid \mathsf{let}_{\mathsf{par}}\ z_1 = {} & \\ & \mathsf{let}_{\epsilon}\ z_2 = \mathsf{read}\, x\ \mathsf{in} \\ & \quad \lambda(z : \mathsf{Nat}).\ z_2 + z \\ \mathsf{in}\ \mathsf{let}_{\epsilon}\ z_3 = \mathsf{read}\, y\ \mathsf{in}\ & z_1\ z_3\end{aligned}$$

It reads two mutable variables x and y (or the reader capabilities or the two mutable states) and adds them together. Note that, for the sake of this example, we assume standard natural numbers as part of the language. Additionally, we assume the initial store to be γ with the value resulted from reading x and y being 1 and 2 respectively. We reduce the reading operation from x in the

first step (which is in the binding term of the parallel let binding), resulting in the following term:

$$\begin{array}{l} \gamma \mid \mathsf{let}_{\mathsf{par}}\ z_1 = \\ \quad \mathsf{let}_{\epsilon}\ z_2 = \boxed{1}\ \mathsf{in} \\ \quad\quad \lambda(z : \mathsf{Nat}).\ z_2 + z \\ \quad \mathsf{in}\ \mathsf{let}_{\epsilon}\ z_3 = \mathsf{read}\ y\ \mathsf{in}\ z_1\ z_3 \end{array}$$

The second step focuses on the body term of the parallel let binding, reducing it to:

$$\begin{array}{l} \gamma \mid \mathsf{let}_{\mathsf{par}}\ z_1 = \\ \quad \mathsf{let}_{\epsilon}\ z_2 = 1\ \mathsf{in} \\ \quad\quad \lambda(z : \mathsf{Nat}).\ z_2 + z \\ \quad \mathsf{in}\ \mathsf{let}_{\epsilon}\ z_3 = \boxed{2}\ \mathsf{in}\ z_1\ z_3 \end{array}$$

This demonstrates the *parallelism* of CSC's reduction: the binding term and the body term of the parallel binding are reduced interleavingly. We continue to evaluate the body term, reducing it into:

$$\begin{array}{l} (\gamma, \boxed{\mathsf{val}\ z_3 = 2}) \mid \mathsf{let}_{\mathsf{par}}\ z_1 = \\ \quad\quad \mathsf{let}_{\epsilon}\ z_2 = 1\ \mathsf{in} \\ \quad\quad\quad \lambda(z : \mathsf{Nat}).\ z_2 + z \\ \quad\quad \mathsf{in}\ \boxed{z_1\ z_3} \end{array}$$

This step uses the (LIFT-LET) rule to lift the value of z_3 to the store context. At this point, we cannot evaluate the body term any further, since it requires the value of z_1, which is not available yet. The reduction of body term is blocked until the binding term is fully reduced, which is similar to awaiting a future. We continue to reduce the binding term, lifting the evaluated value to the store:

$$(\gamma, \mathsf{val}\ z_3 = 2, \boxed{\mathsf{val}\ z_2 = 1}, \boxed{\mathsf{val}\ z_1 = \lambda(z : \mathsf{Nat}).\ z_2 + z}) \mid \boxed{z_1\ z_3}$$

The lifted store binding unblocks the reduction of the body term. This is like the resolution of a future. Finally, we use the (APPLY) rule and reduce the term into:

$$(\gamma, \mathsf{val}\ z_3 = 2, \mathsf{val}\ z_2 = 1, \mathsf{val}\ z_1 = \lambda(z : \mathsf{Nat}).\ z_2 + z) \mid \boxed{3}$$

CSC models mutable variables by keeping the trace. Specifically, there are two kinds of store bindings for mutable variables: $\mathsf{var}\ x := v$ signifies the definition of a new mutable cell along with its initial value; and $\mathsf{set}\ x := v$ tracks an update to x. The value of a mutable variable x is determined by its most recent $\mathsf{set}\ x := v$ binding. If none exists, the initial value is used. The idea is reflected by the following definition of mutable variable lookup.

DEFINITION 3.2 (MUTABLE VARIABLE LOOKUP). *$\gamma(\mathit{var}\,x)$ denotes the result of looking up the mutable variable x in the store γ. It is defined as follows:*

$$\begin{array}{lcll} \gamma(\mathit{var}\,x) & = & v, & \mathit{if}\ \gamma = (\gamma', \mathit{set}\,x := v), \\ \gamma(\mathit{var}\,x) & = & v, & \mathit{if}\ \gamma = (\gamma', \mathit{var}\,x := v), \\ \gamma(\mathit{var}\,x) & = & \gamma'(\mathit{var}\,x), & \mathit{if}\ \gamma = (\gamma', \mathit{var}\,y := v), \\ \gamma(\mathit{var}\,x) & = & \gamma'(\mathit{var}\,x), & \mathit{if}\ \gamma = (\gamma', \mathit{set}\,y := v), \\ \gamma(\mathit{var}\,x) & = & \gamma'(\mathit{var}\,x), & \mathit{if}\ \gamma = (\gamma', \mathit{val}\,y = v). \end{array}$$

The (LIFT-VAR) rule creates a new mutable variable. It first looks up the immutable variable y to obtain the initial value v, then lifts the var $x := v$ binding to the store. The (WRITE) rule updates a mutable variable x to a new value v by creating a set $x := v$ binding in the store. The (READ) rule looks up the value of a mutable variable using the $\gamma(\mathsf{var}\ x)$ function.

An alternative for modelling mutable variables is to update its value in-place when a variable is mutated. The problem is that it complicates the typing of the store, as in-place updates can result in forward dependency between bindings, where earlier bindings depend on later ones. Now we inspect an example:[5]

$$\begin{aligned}&\mathsf{var}\ f = \lambda(x : \mathsf{Int}).x\ \mathsf{in}\\&\mathsf{var}\ g = \lambda(x : \mathsf{Int}).(\mathsf{read}\, f)\ x\ \mathsf{in}\\&\quad f := \lambda(x : \mathsf{Int}).(\mathsf{read}\, g)\ x\end{aligned}$$

If we update the store bindings *in place*, the evaluation of this term will result in such a store:

$$\gamma,\quad \mathsf{var}\ f := \lambda(x : \mathsf{Int}).(\mathsf{read}\, g)\ x,\quad \mathsf{var}\ g := \lambda(x : \mathsf{Int}).(\mathsf{read}\, f)\ x$$

The value of f depends on a later binding, g. In fact, f and g are mutually recursive in this case. The typing of such stores will arguably much more sophisticated in order to model the mutual dependencies between bindings.

By contrast, the trace-based semantics always result in a store with only backward dependency (i.e. later bindings depend on earlier bindings), so that the store can be typed in a standard way. For instance, the trace-based store after the evaluation of the above term is:

$$\gamma,\quad \mathsf{var}\ f := \lambda(x : \mathsf{Int}).x,\quad \mathsf{var}\ g := \lambda(x : \mathsf{Int}).(\mathsf{read}\, f)\ x,\quad \mathsf{set}\ f := \lambda(x : \mathsf{Int}).(\mathsf{read}\, g)\ x$$

4 METATHEORY

We prove the type soundness of CSC through the standard progress and preservation theorems. Additionally, we prove the execution of well-typed programs is data-race-free by establishing the confluence of reduction semantics. The full proof of the theorems in this section can be found in Appendix A.

4.1 Proof Devices

We first introduce the supporting proof devices.

4.1.1 Store Typing. Figure 3 defines the typing of stores and evaluation contexts. $\vdash \gamma \sim \Gamma$ states that the store γ can be typed as a typing context Γ. (ST-VAL) and (ST-VAR) types the immutable and mutable bindings in the store. (ST-VAL) types the value v as capturing a precise capture set $\mathsf{cv}(v)$, and (ST-VAR) introduces the mutable variable binding with the separation degree spanning over the entire context (i.e. being $\mathsf{dom}(\Gamma)$). These treatments type the store as a precise and strong typing context, which eases the proof. The (ST-SET) rule types an update to the variable x by verifying that the mutable variable x is defined in the context and the type of the new value v matches the type of x.

4.1.2 Evaluation Context Typing. CSC additionally introduces the typing of evaluation contexts. $\Gamma \vdash e \sim \Delta$ states that the evaluation context e can be typed as Δ under an existing typing context Γ, where in the metatheory the Γ is always obtained from the typing of a store γ ($\vdash \gamma \sim \Gamma$). The need for this judgment arises from the parallel semantics of the calculus. When reasoning about a

[5]For the simplicity of presentation, the example does not strictly follow the MNF, but the translation into MNF is straightforward.

Store Typing $\vdash \gamma \sim \Gamma$

$$\frac{}{\vdash \cdot \sim \emptyset} \text{(ST-EMPTY)} \qquad \frac{\vdash \gamma \sim \Gamma \qquad \Gamma \vdash v : S}{\vdash (\gamma, \mathsf{var}\ x := v) \sim \Gamma, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S]\,\hat{}\,\{\mathbf{ref}\}} \text{(ST-VAR)}$$

$$\frac{\vdash \gamma \sim \Gamma \qquad \Gamma \vdash v : S\,\hat{}\,\mathsf{cv}(v)}{\vdash (\gamma, \mathsf{val}\ x = v) \sim \Gamma, x :_{\{\}} S\,\hat{}\,\mathsf{cv}(v)} \text{(ST-VAL)} \qquad \frac{\vdash \gamma \sim \Gamma \qquad x :_D \mathsf{Ref}[S]\,\hat{}\,C \in \Gamma \qquad \Gamma \vdash v : S}{\vdash (\gamma, \mathsf{set}\ x := v) \sim \Gamma} \text{(ST-SET)}$$

Evaluation Context Typing $\Gamma \vdash e \sim \Delta$

$$\frac{}{\Gamma \vdash [] \sim \emptyset} \text{(EV-EMPTY)}$$

$$\frac{\Gamma \vdash e \sim \Delta}{\Gamma \vdash \mathsf{let}_m\ x = e\ \mathsf{in}\ s \sim \Delta} \text{(EV-LET-1)} \qquad \frac{\Gamma \vdash s : T \qquad \Gamma, x :_{\{\}} T \vdash e \sim \Delta}{\Gamma \vdash \mathsf{let}_{\mathsf{par}}\ x = s\ \mathsf{in}\ e \sim x :_{\{\}} T, \Delta} \text{(EV-LET-2)}$$

Fig. 3. Store and Evaluation Context Typing

term $e\,[\,s\,]$ under a store γ (typed as $\vdash \gamma \sim \Gamma$), there may be not-yet reduced parallel bindings in e and they can be referred to in s. Therefore, to reason about the focused term s we have to extract and type the bindings from the evaluation context e as well (using the judgment $\Gamma \vdash e \sim \Delta$), so that s is well-typed under Γ, Δ.

In the proof we use $\vdash \gamma; e \sim \Gamma; \Delta$ as a shorthand for $(\vdash \gamma \sim \Gamma) \wedge (\Gamma \vdash e \sim \Delta)$, and $\gamma \vdash t$ denotes $\exists \Gamma, T.(\vdash \gamma \sim \Gamma) \wedge (\Gamma \vdash t : T)$.

4.1.3 Well-Formed Environment. In the metatheory we assume that all environments we deal with are well-formed. An environment Γ is well-formed if all bindings it contains are well-formed in the defining environment, i.e. given $\Gamma_0, x : T$, we have $\Gamma_0 \vdash T$ **wf**. Since well-formedness is implicitly assumed, all the transformations on the environments should preserve well-formedness.

4.1.4 Inertness. We introduce the notion of inertness, which is a property for typing contexts, to reflect the idea that the separation degrees of the bindings are well-grounded. Specifically, the separation degree should either

- be introduced by a mutable variable binding, as a fresh mutable variable can specify an arbitrary separation degree;
- or be derivable from the capture set of the type. In other words, given a binding $x :_D S\,\hat{}\,C$, D is derivable from the capture set C if $C \bowtie D$.

Definition 4.1 (Inert environment). *We say Γ is inert iff $\forall x :_D S\,\hat{}\,C \in \Gamma$ and $\Gamma = \Gamma_1, x :_D S\,\hat{}\,C, \Gamma_2$ for some Γ_1 and Γ_2, one of the following holds:*

- *$C = \{\mathbf{ref}\}$, and $D = dom(\Gamma_1)$,*
- *or $C \cap \{\mathbf{cap}, \mathbf{ref}, \mathbf{rdr}\} = \{\}$, and $\Gamma_1 \vdash D \bowtie C$.*

The following facts can be straightforwardly verified by inspecting the definition of store and evaluation context typing.

Fact 4.1. *$\vdash \gamma \sim \Gamma$ implies that Γ is inert.*

Fact 4.2. *$\Gamma \vdash e \sim \Delta$ implies that Δ is inert.*

4.2 Type Soundness

To establish the type soundness of CSC, we prove the standard preservation and progress theorems [Wright and Felleisen 1994].

Theorem 4.3 (Preservation). *If (i) $\vdash \gamma \sim \Gamma$, (ii) $\Gamma \vdash t : T$, and (iii) $\gamma \mid t \longmapsto \gamma' \mid t'$, then $\exists \Gamma'$ such that (i) $\vdash \gamma' \sim \Gamma'$ and (ii) $\Gamma' \vdash t' : T$.*

Theorem 4.4 (Progress). *If (i) $\vdash \gamma \sim \Gamma$, (ii) $\Gamma \vdash t : T$, then either t is an answer, or $\exists \gamma', t'$ such that $\gamma \mid t \longmapsto \gamma' \mid t'$.*

When proving preservation, we need to deal with the interaction between the evaluation context and the typing of the term. Specifically, given a term $e\,[\,s\,]$, we have to inspect the typing of s under the evaluation context e, and after reducing s to s' we have to plug the reduced term back into the evaluation context e (to obtain $e\,[\,s'\,]$). To analyze the typing of the "hole" inside an evaluation context e, we introduce the notion of *evaluation context inversion* and prove the corresponding lemma for inverting an evaluation context.

Definition 4.2 (Evaluation context inversion). *We say $\Gamma \vdash e \sim \Delta : [U \mathbin{@} s] \Rightarrow T$ iff (i) $\Gamma \vdash e \sim \Delta$, (ii) $\forall s'.\ \Gamma, \Delta \vdash s' : U$ and $\Gamma; \Delta \vdash s' \triangleleft: s$ imply $\Gamma \vdash e\,[\,s'\,] : T$.*

Lemma 4.5 (Inversion of evaluation contexts). *If $\Gamma \vdash e\,[\,s\,] : T$, then $\exists \Delta, U$ such that (i) $\Gamma \vdash e \sim \Delta : [U \mathbin{@} s] \Rightarrow T$, and (ii) $\Gamma, \Delta \vdash s : U$.*

To plug a term s' back, it should not only match the type required by the hole, but also be *fresher* than the term s. In the metatheory, the term s is always the original term focused by the evaluation context e (i.e. $e\,[\,s\,]$ is the term being reduced). A fresher term achieves a larger degree of separation in terms of separation checking. In other words, for all separation checks that s passes, a term s' fresher than s should also pass. This idea is captured by the following definition.

Definition 4.3 (Fresher Term). *The term s is considered fresher than another term t under typing context Γ, Δ (written $\Gamma; \Delta \vdash s \triangleleft: t$), iff given any Δ_1, Δ_2, e, such that $\Delta = \Delta_1, \Delta_2, \Gamma, \Delta_1 \vdash e \sim \Delta_2$ we have $\forall C.\ \Gamma, \Delta_1 \vdash e\,[\,t\,] \bowtie C$ implies $\Gamma, \Delta_1 \vdash e\,[\,s\,] \bowtie C$.*

The freshness of s' is needed when plugging it into a parallel let binding, whose typing rule requires the separation between the binding term and the body. Specifically, given the original term $e\,[\,s\,] = e'\,\big[\,\mathrm{let}_{\mathrm{par}}\ x = s\ \mathrm{in}\ t\,\big]$, to plug a term s' into the hole (i.e. replacing s with s') and preserve the typing, we have to ensure the separation between s' and t,

The lemmas needed for the preservation and progress theorems are standard. We refer the reader to the appendix for the full proofs.

4.3 Data Race Freedom

We prove that CSC statically ensures data race freedom by proving the confluence of its reduction. In other words, despite that the reduction of the binding term and the body in parallel let bindings can be arbitrarily interleaved, the final result of reducing a well-typed term is deterministic and independent of the order of reduction. This means that no data races can occur during the reduction.

Theorem 4.6 (Confluence). *Given two equivalent configurations $\gamma_1 \mid t \cong \gamma_2 \mid t$, if (1) $\gamma_1 \vdash t$ and $\gamma_2 \vdash t$; (2) $\gamma_1 \mid t \longmapsto^* \gamma_1' \mid t_1$; and (3) $\gamma_2 \mid t \longmapsto^* \gamma_2' \mid t_2$, then there exists $\gamma_1'', t', \gamma_2''$ such that (1) $\gamma_1' \mid t_1 \longmapsto^* \gamma_1'' \mid t'$, (2) $\gamma_2' \mid t_2 \longmapsto^* \gamma_2'' \mid t'$, and (3) $\gamma_1'' \mid t' \cong \gamma_2'' \mid t'$.*

Theorem 4.7 (Uniqueness of Answer). *For any t, if (i) $\gamma \vdash t$, (ii) $\gamma \mid t \longmapsto^* \gamma_1 \mid a_1$ and (iii) $\gamma \mid t \longmapsto^* \gamma_2 \mid a_2$ then $a_1 = a_2$ and $\gamma_1 \cong \gamma_2$.*

Theorem 4.7 is a corollary of Theorem 4.6. $\gamma \mid t \longmapsto^* \gamma' \mid t'$ denotes the transitive and reflexive closure of the small step reduction $\gamma \mid t \longmapsto \gamma' \mid t'$. $\gamma \vdash t$ is a shorthand for $\exists \Gamma, T$ such that $\vdash \gamma \sim \Gamma$ and $\Gamma \vdash t : T$. In the theorem, we use the equivalence between evaluation configurations $\gamma_1 \mid t \cong \gamma_2 \mid t$ instead of the equality. $\gamma_1 \mid t_1 \cong \gamma_2 \mid t_2$ is a shorthand for the conjunction of the equivalence between two stores $\gamma_1 \cong \gamma_2$ and the equality between two terms $t_1 = t_2$. Two equivalent stores $\gamma_1 \cong \gamma_2$ have their bindings ordered differently but represent exactly the same state of immutable and mutable variables, in the sense that looking up any variable in both stores yields the same result. In other words, though they have different *internal* representations, the two stores are *externally* indistinguishable. For instance, $\mathsf{var}\ x := 0, \mathsf{var}\ y := 0, \mathsf{set}\ x := 10, \mathsf{set}\ y := 11$ and $\mathsf{var}\ y := 0, \mathsf{var}\ x := 0, \mathsf{set}\ y := 11, \mathsf{set}\ x := 10$ are not equal but equivalent stores. The following definition formalizes this idea.

Definition 4.4 (Equivalent stores). *We say two stores γ_1 and γ_2 are equivalent, written $\gamma_1 \cong \gamma_2$, iff*

(1) γ_2 is permuted from γ_1;
(2) $\forall x \in bvar(\gamma_1), \gamma_1(var\ x) = \gamma_2(var\ x)$.

Definition 4.5 (Equivalent configurations). *We say two configurations $\gamma_1 \mid t_1$ and $\gamma_2 \mid t_2$ are equivalent iff $\gamma_1 \cong \gamma_2$ and $t_1 = t_2$.*

We need the notion of store equivalence because two different paths of reduction will generally not converge to having the same store (as store bindings are lifted in different orders), but they will converge to having equivalent stores.

We take the standard approach towards confluence by proving the diamond property [Church and Rosser 1936]. The following theorem establishes the diamond property for a single step of reduction.

Theorem 4.8 (Diamond property of reduction). *Given two equivalent configurations $\gamma_1 \mid t \equiv \gamma_2 \mid t$, if (1) $\gamma_1 \vdash t$ and $\gamma_2 \vdash t$; (2) $\gamma_1 \mid t \longmapsto \gamma_1' \mid t_1$; and (3) $\gamma_2 \mid t \longmapsto \gamma_2' \mid t_2$, then either $t_1 = t_2$, or there exists $\gamma_1'', t', \gamma_2''$ such that (1) $\gamma_1' \mid t_1 \longmapsto \gamma_1'' \mid t'$, (2) $\gamma_2' \mid t_2 \longmapsto \gamma_2'' \mid t'$, and (3) $\gamma_1'' \mid t' \equiv \gamma_2'' \mid t'$.*

This is a prerequisite for proving Theorem 4.6, which is essentially the diamond property for multi-step reduction.

The following two propositions play an important role when establishing the diamond property. They state that as long as the two references are statically known to be separated, they do not alias the same state and therefore cannot introduce data races.

Proposition 4.9. *If (i) $\Gamma \vdash \{x\} \bowtie \{y\}$, (ii) $x : S_1 \wedge \{\mathbf{ref}\} \in \Gamma$ and (iii) $y : S_2 \wedge \{\mathbf{ref}\} \in \Gamma$, then we have $x \neq y$.*

Proposition 4.10. *If (i) $\Gamma \vdash \{x\} \bowtie \{y\}$, (ii) $x : Rdr[S_1] \wedge \{z\} \in \Gamma$ and (iii) $y : S_2 \wedge \{\mathbf{ref}\} \in \Gamma$, then $z \neq y$.*

5 DISCUSSION

We discuss backward compatibility guarantees of CSC, one of its primary design goals. We also discuss how CSC can handle linked object graphs.

5.1 Backward Compatibility

Backward compatibility means that if a program is well-typed in a pre-CSC system, it remains typeable with the introduction of CSC. This goal is realized by two key features of CSC:

- *Separation checking is opt-in.* Since separation degrees can be set to empty by default, and empty separation degrees does not require any separation checking, we can avoid affecting existing codebases upon the introduction of CSC.
- *The system is permissive to local aliasing*. Local aliases are always allowed, yet tracked, as long as they do not violate separation degree requirements.

Another essential aspect of backward compatibility is the way to model complex object graphs, which revolves around allowing objects to be encapsulated and aggregated. The literature on this topic is vast. Examples include reference capabilities [Castegren and Wrigstad 2016; Gordon et al. 2012], ownership types [Boyapati and Rinard 2001; Clarke et al. 1998], and balloon types [Servetto et al. 2013]. There are diverse possibilities for extending CSC with the support for modelling complex object graphs, each with its own challenges and trade-offs. Instead of prematurely committing to any particular approach, CSC is a minimal core system, a lean foundation for our approach to separation which remains open to future extensions.

5.2 Handling Complex Object Graphs

Although it is a minimal formalisation, CSC is capable of typechecking certain linked object graphs. We demonstrate how CSC typechecks singly-linked lists of mutable references. We also show that CSC can model simple mutable and cyclic object graphs, assuming a simple class extension.

5.2.1 Linked Lists. We demonstrate an example of typechecking linked object graphs in CSC: singly linked lists. The following snippet shows how a church-encoding of a singly-linked list of mutable integer references:

```
type IntList[C] = [R] ->^C (init: R) ->^C (op: (x: Ref[Int]^) => R => R) ->^C R
def nil: (IntList[{}]) = [R] => init => op => init
def cons(x: Ref[Int]^, xs: IntList[{cap}]): (IntList[{x, xs}]) =
  [R] => init => op => op(x, xs(init)(op))
```

Recall that `T^` is a syntactic sugar for `T^{cap}`. Besides, `IntList[C]` is an abbreviation which expands to the RHS of the `type` definition.

We can define the following operations on the list:

```
def sum(xs: IntList[{cap}]): Int =
  xs(0)((x, acc) => x.reader.get + acc)
def inc(xs: IntList[{cap}]): Int =
  xs(unit)((x, acc) => x.set(x.reader.get + 1); unit)
```

This example also shows that the separation approach proposed in CSC is opt-in by its nature. This program is how users could write a linked list of integer references in a system before separation checking is introduced, and remains as it is and well-typed with separation checking turned on. The user is not forced to reason about the ownership or uniqueness of the integer references stored in the linked list or of the linked list itself.

When the separation between linked lists matters, the user can ask the type system to enforce it by adding **sep** annotations, like the following:

```
def sumPar(xs1: IntList[{cap}], sep{xs1} xs2: IntList[{cap}]) =
  letpar s1 = sum(xs1) in
    val s2 = sum(xs2)
    s1 + s2
```

The program will not typecheck without the **sep** annotation on the argument `xs2`.

Furthermore, CSC is capable of encoding a list of mutually separated integer references.

```
type SepIntList[C] = [R] ->^C (init: R) ->^C (op: (x: Ref[Int]^) -> R -> R) ->^C R
def nil: (SepIntList[{}]) = [R] => init => op => init
def cons(x: Ref[Int]^, sep{x} xs: SepIntList[{cap}]): (IntList[{x, xs}]) =
  [R] => init => op =>
    letpar acc = xs(init)(op) in
      op(x, acc)
```

Compared to `IntList`, the `cons` function requires the tail of the list, passed in as the argument `xs`, to be separated from the head `x`. The separation between the list elements is then made use of in the `cons` function, which performs the fold of the list in parallel.

One limitation of this encoding is that the `op` argument in the type definition of `SepIntList` has to be pure. Otherwise the `cons` function, which concurrently calls `op` and accesses `x` and `xs`, will not typecheck.

5.2.2 Cyclic and Mutable Object Graphs. In the following, we demonstrate that the system is capable of typechecking cyclic and mutable object graphs.

```
class Node:
  val data: Ref[Int]^{ref}
  val next: Ref[□ Node^{ref}]^{ref}
```

This defines a node in the object graph. It assumes a class extension to the calculus. Studying the interaction between the separation calculus and classes is an important future direction, and is a natural next step when applying our approach to a language like Scala. But it is left as future work and the example is a proof-of-concept. Each node contains a mutable integer and a pointer to another node. The pointer is mutable and can be updated to point to a different node. The following example is a simple cyclic object graph that consists of two nodes pointing to each other:

```
// (1) Create payloads
val a: Ref[Int]^{ref} = new Ref(42)
val b: Ref[Int]^{ref} = new Ref(42)
// (2) Allocate references to nodes
val pa: Ref[□ Node^{ref}]^{ref} = new Ref(null)
val pb: Ref[□ Node^{ref}]^{ref} = new Ref(null)
// (3) Create the two nodes
val na: Node^{a, pa} = Node(a, pa)
val nb: Node^{b, pb} = Node(b, pb)
// (4) Let them point to each other
pa.set(□ nb)
pb.set(□ na)
```

The following code demonstrates a simple usage of this object graph. It follows the pointer `next` of the node `na` and updates the mutable integer of the node pointed to:

```
// (1) Read the `next' field of `na'
val boxedNode: □ Node^{ref} = na.next.reader.get
// (2) Unbox the reference
val node: Node^{ref} = {ref} ⟜ boxedNode
// (3) Mutate the integer payload
node.data.set(0)
```

6 RELATED WORK

Linearity and Uniqueness. There is a rich literature on linear and uniqueness types [Barendsen and Smetsers 1996; Boyland 2001; Fähndrich and Deline 2002; Marshall et al. 2022; Wadler 1990]. Both are type systems that enforces strict alias prevention invariants. A linear value should be consumed *exactly once*, and is unusable after consumption. This requirement makes linear types a perfect match for checking resource protocols. For instance, a file handle, once opened, should be closed exactly once and discarded afterwards: forgetting to close it leads to a resource leak, using it after closing it results in a use-after-free error, and closing it twice causes undefined behaviors. Classical linear types enforces a strict alias-free invariant: since linear values can only be used once, they are guaranteed to be un-aliased. Uniqueness types similarly guarantees an alias-free invariant for unique values. Linearity plays an important role in Rust's fearless concurrency [Klabnik and Nichols 2018; Weiss et al. 2019]. Rust mandates that a thread takes ownership of whatever it captures, and the linear ownership system effectively precludes potential aliases to the data captured by a thread. It is widely recognized that the alias restriction of these systems is too strict for many practical use cases, and there have been attempts to relax the restriction [Clarke and Wrigstad 2003; Fähndrich and Deline 2002; Noble et al. 2022; Odersky 1992]. However, the alias-free invariant still lies at the heart of these systems and additional annotations or code refactoring are required to adopt them into existing code bases.

Ownership types. Ownership types [Clarke et al. 1998] provides a type system that is capable of enforcing *encapsulation*, or *representation containment*. It precludes aliases to data owned by an object that cross the boundary of the object. Viewing a thread as an object, ownership types can be used to model thread-local data that cannot be aliased by other threads [Boyapati and Rinard 2001] and can thus be accessed freely without introducing data races. Ownership types can be extended with uniqueness [Boyapati and Rinard 2001; Hogg 1991; Servetto et al. 2013] to enforce expressive alias prevention invariants. However, ownership types place restrictions on inter-object aliasing, which often requires rethinking the organization of program data when they are adopted into existing code bases.

Reference capabilities. Reference capabilities [Castegren and Wrigstad 2016; Gordon et al. 2012] tags object references with modes. The type system enforces different alias invariants for different modes. For instance, in κ [Castegren and Wrigstad 2016], a `linear` reference is globally unique, while a `thread` reference allows aliases within the same thread, and precludes sharing across threads.

Fractional permissions. Fractional permissions allow temporarily share a mutable variable immutably [Boyland 2010]. In this system, accessing mutable variables requires *permissions*. Once created, a mutable variable is associated with a *full permission*, which authorizes write access to the variable. One can split the full permission into multiple read permissions to perform read operations on the variable in a shared manner. Afterwards, all read permissions have to be relinquished to regain the full permission so that the variable can be mutated again. This reflects the multiple-reader-single-writer model in concurrent programming. Similar ideas can be found in other systems [Crary et al. 1999; Fähndrich and Deline 2002]. In CSC, a reader capability can be viewed as a *fraction* of the full capability. However, a reader capability can co-exist with a full capability and be used together sequentially. Only the parallel usage of reader capabilities and the full capability of a variable is prohibited.

Reachability types. The recent proposal, reachability types [Bao et al. 2021; Wei et al. 2023], tracks lifetimes and sharing with reachability qualifiers in types. The reachability qualifier in a type indicates the set of variables that is reachable from the value of that type. The first version of

reachability types, System λ^*, is proposed by Bao et al.. As long as the reachability qualifiers in the types of two values are disjoint, the two values are guaranteed to reach disjoint objects. However, λ^* is monomorphic and it turns out that a naive polymorphism extension makes the system unsound [Wei et al. 2023]. Later, Wei et al. proposes Polymorphic Reachability Types (System $\lambda^{\blacklozenge}$) to address that limitation. $\lambda^{\blacklozenge}$ inherits the reachability qualifiers, and is polymorphic over both types and qualifiers. Qualifiers and capture sets are similar in spirit. Both can be viewed as a alias tracking device. The diamond $\blacklozenge$ in $\lambda^{\blacklozenge}$ resembles the universal capability **cap** in CSC. When appearing in a capture set, **cap** stands for an arbitrary set of variables aliased by a value, while $\blacklozenge$ indicates an arbitrary set of variables that are reachable from a value and separated from the current context. To enforce the *separation* invariant of $\blacklozenge$, $\lambda^{\blacklozenge}$ has a dedicated application rule for functions that take a $\blacklozenge$-qualified parameter: the separation between the function and the argument is checked by that rule. The analogy of a $\blacklozenge$-qualified argument in CSC would be a parameter capturing **cap** with a separation degree being the captured variables of the function body:

$$\lambda(z :_{\mathrm{cv}(t)} S \hat{}\{\mathbf{cap}\}).\, t$$

When type-checking the application of this function, the separation between the function body t and the argument will be established. The function is polymorphic over the aliasing of the parameter, and the parameter z is assumed separated from the whole observable context of the function. Compared to CSC, reachability types enforces the separation between arguments and function bodies by default, and permits aliases with explicit annotations. Therefore, considerable annotations are required to allow for established aliasing patterns when reachability types are introduced to existing languages. Additionally, since data races are not a central concern of reachability types, immutable sharing of mutable state is not modelled by the system.

Flexibility. There have been lots of research interest in the flexibility of alias prevention systems. The *control-as-you-need* principle of CSC resembles that of syntactic control of interference (SCI) [O'Hearn et al. 1999; Reynolds 1978]: interference is possible, but should be syntactically detectable. The passive expressions in SCI are similar to an expression in CSC whose captured variables are upper bounded by **rdr**. However, alias tracking in SCI is not as expressive as that provided by capturing types in CSC: in SCI, distinct variables are always non-interfering. This is equivalent to having the largest separation degree (which is, all other variables defined in the context) for every variable in CSC. Recently, Milano et al. proposes a flexible type system for alias prevention. Its key concept is *tempered domination*: the proposed `iso` reference by default dominates its reachable object graph, which means that it is the unique reference to that graph. This uniqueness requirement can be relaxed by the *focus* mechanism, which allows objects reachable from an `iso` reference to be aliased as long as the aliasing is tracked in the type system. Their system naturally expresses interlinked data structures, like doubly-linked lists, and operations on them. However, the domination property of `iso` references still by default imposes restrictions on heap structures, and therefore requires code refactoring when adopted into existing code bases. Besides, their system models message-passing concurrency, where reachable object graphs between threads are always disjoint, while CSC models shared-memory concurrency, where mutable state can be immutably shared across threads. Finally, unlike CSC, which originates from System $F_{<:}$ and is thus close to a functional language, Milano et al.'s system studies an imperative language.

7 CONCLUSION

In this paper, we have developped CSC, a calculus for static data-race-free concurrency. It addresses the backward-compatibility problem: it is expressive enough to prevent race conditions statically, but is also flexible to accommodate existing programming patterns. We envision this promotes the

adoption of static race freedom to existing code bases by easing the migration of existing code. It achieves this flexibility by following a control-as-you-need principle. Aliases are allowed by default, but are tracked in the types. When data races are a concern, the tracked aliases are controlled and data-race-prone aliasing patterns are rejected. In the metatheory, we establish the type soundness of CSC through the standard progress and preservation theorems. Furthermore, we prove that the reduction of well-typed programs is *confluent*, which formally justifies the static race freedom guarantee of CSC.

ACKNOWLEDGMENTS

This research was partially funded by Swiss National Science Foundation grant TMAG-2_209506/1. We thank Ondřej Lhoták, Jonathan Brachthäuser and Edward Lee for their valuable feedbacks during the discussions of this work. We thank the anonymous reviewers for their insightful comments and suggestions.

REFERENCES

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World*.

Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proceedings of the ACM on Programming Languages* 5 (2021), 1 – 32.

Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science* 6 (1996), 579 – 612.

Chandrasekhar Boyapati and Martin C. Rinard. 2001. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*. 56–69.

John Tang Boyland. 2001. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience* 31 (2001).

John Tang Boyland. 2010. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.* 32 (2010), 22:1–22:33.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages* 4 (2020), 1 – 30.

Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *European Conference on Object-Oriented Programming*.

Alonzo Church and J. Barkley Rosser. 1936. Some properties of conversion. *Trans. Amer. Math. Soc.* 39 (1936), 472–482.

Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *European Conference on Object-Oriented Programming*.

David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. 48–64.

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (2015).

Karl Crary, David Walker, and Greg Morrisett. 1999. Typed memory management in a calculus of capabilities. In *ACM-SIGACT Symposium on Principles of Programming Languages*.

Manuel Fähndrich and Robert A Deline. 2002. Adoption and focus: practical linear types for imperative programming. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*.

Kasra Ferdowsi. 2023. The Usability of Advanced Type Systems: Rust as a Case Study. *ArXiv* abs/2301.02308 (2023).

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. The MIT Press.

Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. In *OOPSLA*. 21–40.

J. Hogg. 1991. Islands: aliasing protection in object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*.

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press, USA.

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (2017).

Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership Is Theft: Experiences Building an Embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. ACM, Monterey California, 21–26. https://doi.org/10.1145/2818302.

2818306
Daniel Marshall, Michael Vollmer, and Dominic A. Orchard. 2022. Linearity and Uniqueness: An Entente Cordiale. In *European Symposium on Programming*.
Mae Milano, Joshua Turcotti, and Andrew C. Myers. 2022. A flexible type system for fearless concurrency. *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (2022).
Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph. D. Dissertation. USA. Advisor(s) Shapiro, Jonathan S. AAI3245526.
James Noble, Julian Mackay, and Tobias Wrigstad. 2022. Rusty Links in Local Chains. *ArXiv* abs/2205.00795 (2022).
Martin Odersky. 1992. Observers for Linear Types. In *European Symposium on Programming*.
Martin Odersky, Aleksander Boruch-Gruszecki, Edward Lee, Jonathan Immanuel Brachthäuser, and Ondrej Lhoták. 2022. Scoped Capabilities for Polymorphic Effects. *ArXiv* abs/2207.03402 (2022).
Peter W. O'Hearn, John Power, Robert D. Tennent, and Makoto Takeyama. 1999. Syntactic control of interference revisited. In *Mathematical Foundations of Programming Semantics*.
Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. 2022. Implementation Strategies for Mutable Value Semantics. *J. Object Technol.* 21 (2022), 2:1–11.
Marianna Rapoport and Ondrej Lhoták. 2019. A path to DOT: formalizing fully path-dependent types. *Proceedings of the ACM on Programming Languages* 3 (2019), 1 – 29.
John C. Reynolds. 1978. Syntactic control of interference. *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1978).
Scala. 2022. The Scala 3 compiler, also known as Dotty. https://dotty.epfl.ch
Marco Servetto, David J. Pearce, Lindsay J. Groves, and Alex Potanin. 2013. Balloon Types for Safe Parallelisation over Arbitrary Object Graphs.
Philip Wadler. 1990. Linear Types can Change the World!. In *Programming Concepts and Methods*.
Guannan Wei, Oliver Bracevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2023. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *ArXiv* abs/2307.13844 (2023).
Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal J. Ahmed. 2019. Oxide: The Essence of Rust. *ArXiv* abs/1903.00982 (2019).
Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115 (1994), 38–94.

A PROOF

Throughout the proof, we follow the Barendregt convention where all the bound variables are distinct.

A.1 Proof Devices

In addition to the proof devices introduced in Section 4, the following devices are used in the metatheory as well. Now we define them and explain their usage.

A.1.1 Binding Depth. Binding depth is the "index" of a binding in the context. It helps us to define the induction schemes on capture sets.

Definition A.1 (Depth). *We define the depth of the variable x in an environment Γ as the index of x in Γ, i.e.*

$$\begin{aligned} depth_{\Gamma,x:T}(x) &= |\Gamma| \\ depth_{\Gamma,x:T}(y) &= depth_{\Gamma}(x) \end{aligned}$$

Note that $\mathrm{depth}_\Gamma(x)$ is a partial function: it is only defined on the domain of Γ. Notably, it is not defined on the special root capabilities **cap**, **rdr** and **ref**.

Definition A.2 (Depth of Capture Set). *We define the depth of a capture set C, written $[\![C]\!]$, as the maximal depth of the variables in C, i.e.*

$$[\![C]\!] = \max_{x \in C \setminus \{\mathbf{cap},\mathbf{rdr},\mathbf{ref}\}} depth_{\Gamma}(x) .$$

Specially, we let $[\![C]\!] = -1$ *if* $C \setminus \{\textbf{\textit{cap}}, \textbf{\textit{rdr}}\}$ *is empty.*

In Lemma A.27, the induction is carried out on a lexical order of the depth of the capture set and the height of the derivation tree.

A.1.2 Auxilliary Judgment for is-reader. The (SC-READER) subcapturing rule uses the is-reader$_\Gamma$ x judgment, which itself depends on subtyping. This results in the subcapturing and the subtyping rules being mutually dependent, which complicates the induction scheme on this two judgments.

To disentangle subcapturing and subtyping rules, we define the following judgment which is equivalent to the check implemented by is-reader, but eliminates the dependency on subtyping.

$$\Gamma \vdash \mathsf{Rdr}[S]\,\hat{}\,C \uparrow \mathsf{Rdr} \quad \text{(RD-READER)} \qquad \frac{X <: S \in \Gamma \qquad \Gamma \vdash S\,\hat{}\,C \uparrow \mathsf{Rdr}}{\Gamma \vdash X\,\hat{}\,C \uparrow \mathsf{Rdr}} \text{(RD-TVAR)}$$

Later, we will prove the equivalence between $\Gamma \vdash T \uparrow \mathsf{Rdr}$ and $\exists C, S.\Gamma \vdash T <: \mathsf{Rdr}[S]\,\hat{}\,C$.

A.2 Soundness

A.2.1 Properties of Subcapturing.

Lemma A.1 (Decomposition of subcapturing). *$\Gamma \vdash C_1 <: C_2$ implies $\forall x \in C_1.\Gamma \vdash \{x\} <: C_2$.*

Proof. By induction on the subcapture derivation.

Case (SC-VAR), (SC-ELEM), (SC-REF-CAP), (SC-RDR-CAP) and (SC-READER). The proof is concluded trivially since in these cases C_1 is a singleton set.

Case (SC-SET). This case follows directly from the preconditions.

Case (SC-TRANS). Then $\Gamma \vdash C_1 <: C$ and $\Gamma \vdash C <: C_2$. By using the IH, we have $\overline{\Gamma \vdash \{x\} <: C}^{x \in C_1}$, from which we can conclude by using the (SC-TRANS) rule repeatedly. □

Lemma A.2 (Reflexivity of subcapturing). *$\Gamma \vdash C <: C$.*

Proof. We begin by showing that $\overline{\Gamma \vdash \{x\} <: C}^{x \in C}$ using the (SC-ELEM) rule. Afterwards, we may conclude this case by the (SC-SET) rule. □

Lemma A.3 (Set inclusion implies subcapturing). *$C_1 \subseteq C_2$ implies $\Gamma \vdash C_1 <: C_2$.*

Proof. By repeated (SC-ELEM) and (SC-SET). □

Lemma A.4. *$\Gamma \vdash C_1 <: C$ and $\Gamma \vdash C_2 <: C$ implies $\Gamma \vdash C_1 \cup C_2 <: C$.*

Proof. We begin by showing that $\overline{\Gamma \vdash \{x\} <: C}^{x \in C_1}$ and $\overline{\Gamma \vdash \{x\} <: C}^{x \in C_2}$ using Lemma A.1. Now we can conclude this case by applying the (SC-SET) rule. □

Lemma A.5. *$\Gamma \vdash C <: C_1$, implies $\Gamma \vdash C <: C_1 \cup C_2$.*

Proof. By induction on the subcapture derivation.

Case (SC-SET). Then $C = \{x\}$ and $x \in C_1$. We have $x \in C_1 \cup C_2$ and conclude this case by applying the (SC-SET) rule again.

Other cases. By IH and the same rule. □

Corollary A.6. *$\Gamma \vdash C_1 <: C_2$ and $\Gamma \vdash D_1 <: D_2$ implies $\Gamma \vdash C_1 \cup D_1 <: C_2 \cup D_2$.*

Lemma A.7 (Capture set is irrelevant in reader checking). *If $\Gamma \vdash S\,\hat{}\,C \uparrow Rdr$ then $\Gamma \vdash S\,\hat{}\,C' \uparrow Rdr$.*

Proof. By straightforward induction on the derivation. In the (RD-READER) case we conclude from the premise immediately. In the (RD-TVAR) case we conclude by the IH and the same rule. □

Lemma A.8 (Subtyping preserves reader checking). *If* $\Gamma \vdash T \uparrow \mathit{Rdr}$ *and* $\Gamma \vdash T' <: T$ *then* $\Gamma \vdash T' \uparrow \mathit{Rdr}$.

Proof. By induction on the subtyping derivation.
Case (REFL). Immediate.
Case (CAPT). By the IH and Lemma A.7.
Case (TRANS). By repeated application of the IH.
Case (TVAR). Then $T' = X$ and $X <: T \in \Gamma$. We conclude by using the (RD-TVAR) rule. □

Lemma A.9 (Equivalence between reader checking). $\Gamma \vdash T \uparrow \mathit{Rdr}$ *iff* $\exists C, S.\ \Gamma \vdash T <: \mathit{Rdr}[S] \wedge C$.

Proof. We prove the two directions in the equivalence respectively.
(⇒): Proceed the proof by induction on the derivation. In the (RD-READER) case, we conclude immediately by the reflexivity of subtyping. In the (RD-TVAR) case, we conclude by the IH and the (TVAR) rule.
(⇐). By induction on the subtyping derivation. In the (REFL) case we conclude immediately by the (RD-READER) rule. In the (TRANS) case, we have $\Gamma \vdash T <: T'$ and $\Gamma \vdash T' <: C\ \mathsf{Rdr}[S]$ for some T'. We first use the IH to show that $\Gamma \vdash T' \uparrow \mathsf{Rdr}$. Then we invoke Lemma A.8 to conclude this case. Finally, the (TVAR) case can be concluded immediately using the (RD-TVAR) rule. □

Lemma A.10 (Reader checking strengthening). *Given* $\Gamma = \Gamma_1, \Delta, \Gamma_2$, *if* $\Gamma' = \Gamma_1, \Gamma_2$ *is still well-formed,* $\Gamma \vdash T \uparrow \mathit{Rdr}$, *and* T *is well-formed in* Γ', *then* $\Gamma' \vdash T \uparrow \mathit{Rdr}$.

Proof. By straightforward induction on the derivation. In the (RD-READER) case we conclude immediately using the same rule. In the (RD-TVAR) case, we have $T = X \wedge C$, $X <: R \in \Gamma$, and $\Gamma \vdash R \uparrow \mathsf{Rdr}$. By the well-formedness of Γ' we can show that R is well-formed in Γ'. Then we conclude by the IH and the (RD-TVAR) rule. □

Corollary A.11 (is-reader strengthening). *Given* $\Gamma = \Gamma_1, \Delta, \Gamma_2$, *if* $\Gamma' = \Gamma_1, \Gamma_2$ *is still well-formed,* $\Gamma \vdash T \uparrow \mathit{Rdr}$, *and* $x \in \mathit{dom}(\Gamma')$, *then* $\mathit{is\text{-}reader}_{\Gamma'}\ x$.

Lemma A.12 (Subcapture strengthening). *Given* $\Gamma = \Gamma_1, \Delta, \Gamma_2$, *if* $\Gamma' = \Gamma_1, \Gamma_2$ *is still well-formed, and* $\Gamma \vdash C_1 <: C_2$, *then* $\Gamma' \vdash C_1 \setminus \mathit{dom}(\Delta) <: C_2 \setminus \mathit{dom}(\Delta)$.

Proof. By induction on the subcapture derivation.
Case (SC-TRANS). Then $\Gamma \vdash C_1 <: C$ and $\Gamma \vdash C <: C_2$ for some C. By the IH we can show that $\Gamma' \vdash C_1 \setminus \mathrm{dom}(\Delta) <: C \setminus \mathrm{dom}(\Delta)$ and $\Gamma' \vdash C \setminus \mathrm{dom}(\Delta) <: C_2 \setminus \mathrm{dom}(\Delta)$. Hence we conclude using the (SC-TRANS) rule.
Case (SC-VAR). Then $C_1 = \{x\}$, and $x :_D S \wedge C \in \Gamma$. Proceed by a case analysis on whether x is bound in Δ.

- If $x \in \mathrm{dom}(\Delta)$, we have $C_1 \setminus \mathrm{dom}(\Delta) = \varnothing$ and can conclude this case by (SC-SET).
- If $x \notin \mathrm{dom}(\Delta)$, by the well-formedness of Γ' we have $C \cap \mathrm{dom}(\Delta) = \varnothing$, which implies that $C \setminus \mathrm{dom}(\Delta) = C$. Note that $\{x\} \setminus \mathrm{dom}(\Delta) = \{x\}$. We can therefore apply the (SC-VAR) rule to conclude.

Case (SC-ELEM). Then $C_1 = \{x\}$ and $x \in C_2$. Again we proceed by a case analysis on whether x is bound in Δ.

- If $x \in \mathrm{dom}(\Delta)$, we have $C_1 \setminus \mathrm{dom}(\Delta) = \varnothing$ and thus conclude this case using the (SC-SET) rule.

– Otherwise if $x \notin \mathrm{dom}(\Delta)$, we have $x \in C_2 \setminus \mathrm{dom}(\Delta)$ and can conclude this case by the (SC-ELEM) rule.

Case (SC-SET). We conclude by repeated IH and the same rule.

Case (SC-RDR-CAP). We conclude immediately using the same rule since $\{\mathbf{rdr}\} \setminus \mathrm{dom}(\Delta) = \{\mathbf{rdr}\}$ and $\{\mathbf{cap}\} \setminus \mathrm{dom}(\Delta) = \{\mathbf{cap}\}$.

Case (SC-REF-CAP). Analogous to the case for (SC-RDR-CAP).

Case (SC-READER). Then $C_1 = \{x\}$, $\text{is-reader}_{\Gamma, x :_D P, \Delta}\ x$, and $C_2 = \{\mathbf{rdr}\}$. If $x \in \mathrm{dom}(\Delta)$, then $C_1 \setminus \mathrm{dom}(\Delta) = \{\}$ and we can conclude directly. Otherwise, if $x \notin \mathrm{dom}(\Delta)$, we can show that $x \in \mathrm{dom}(\Gamma')$, and then use Corollary A.11 to conclude to show that $\text{is-reader}_{\Gamma'}\ x$. Note that $\{\mathbf{rdr}\} \setminus \mathrm{dom}(\Delta) = \{\mathbf{rdr}\}$. Now we can conclude this case by the (SC-READER) rule. □

A.2.2 *Properties of Typing and Subtyping.*

Lemma A.13 (Subtype inversion: type variable). *If $\Gamma \vdash U <: X \wedge C$, then $U = Y \wedge C'$ for some C', Y, such that $\Gamma \vdash C' <: C$, and $\Gamma \vdash Y <: X$.*

Proof. By induction on the subtype derivation, wherein only the following cases are possible.

Case (REFL). Immediate.

Case (TVAR). Then $U = Y$, $Y <: X \in \Gamma$, and $C = \{\}$. Now we conclude by applying the (TVAR) rule again.

Case (TRANS). Then $\Gamma \vdash U <: U'$ and $\Gamma \vdash U' <: X \wedge C$ for some U'. By IH, we can first show that $U' = Y \wedge C'$ for some C' and Y. Now, we can invoke IH on the derivation $\Gamma \vdash U <: Y \wedge C'$ to show that $U = Z \wedge C''$, $\Gamma \vdash C'' <: C'$ and $\Gamma \vdash Z <: Y$. Finally we conclude by the transitivity of both subcapturing and subtyping.

Case (CAPT). Then $U = S \wedge C'$ for some C, S, $\Gamma \vdash C' <: C$, and $\Gamma \vdash S <: X$. Now we invoke the IH To show that $S = Y \wedge C''$ for some Y, where $C'' = \{\}$ (note that we consider S to be equivalent to a capturing type with an empty capture set), and $\Gamma \vdash Y <: X$. This case is therefore concluded. □

Lemma A.14 (Subtype inversion: mutable reference). *If $\Gamma \vdash U <: \mathit{Ref}[S] \wedge C$, then either (i) U is of the form $X \wedge C'$, $\Gamma \vdash C' <: C$ and $\Gamma \vdash X <: \mathit{Ref}[S]$, or (ii) U is of the form $\mathit{Ref}[S] \wedge C'$, and $\Gamma \vdash C' <: C$.*

Proof. By induction on the subtype derivation. wherein only the following cases apply.

Case (REFL). Then $U = \mathrm{Ref}[S] \wedge C$. This case is concluded immediately.

Case (TVAR). Then $U = X$ and $X <: \mathrm{Ref}[S] \in \Gamma$. We conclude this case by the (TVAR) rule.

Case (CAPT). By IH. □

Lemma A.15 (Subtype inversion: reader). *If $\Gamma \vdash U <: \mathit{Rdr}[S] \wedge C$, then either (i) U in the form of $X \wedge C'$ where $\Gamma \vdash C' <: C$ and $\Gamma \vdash X <: \mathit{Rdr}[S]$, or (ii) U is in the form of $C'\ \mathit{Rdr}[S]$, where $\Gamma \vdash C' <: C$.*

Proof. Analogous to the proof of Lemma A.14. □

Lemma A.16 (Subtype inversion: term abstraction). *If $\Gamma \vdash P <: \forall(x :_D U)T \wedge C$, then either (i) P is of the form $X \wedge C'$, $\Gamma \vdash C' <: C$ and $\Gamma \vdash X <: \forall(x :_D U)T$, or (ii) P is of the form $\forall(x :_D U')T' \wedge C'$ such that $\Gamma \vdash C' <: C$, $\Gamma \vdash U <: U'$, and $\Gamma, x :_D U' \vdash T' <: T$.*

Proof. By induction on the subtype derivation.

Case (REFL). Then $U = \forall(x :_D U)T \wedge C$. We conclude immediately by the reflexivity of subcapture and subtyping.

Case (TVAR). Then $P = X$, $C = \{\}$, and $X <: \forall(x :_D U)T$. Then we conclude immediately.

Case (FUN). Then $P = \forall(x :_D U')T' \wedge C'$ and we conclude from the preconditions.

Case (TRANS). Then $\Gamma \vdash P <: P'$ and $\Gamma \vdash P' <: C\,\forall(x :_D U)T$ for some P'. By IH we can show that P' is either of the form $X \wedge C'$ such that $\Gamma \vdash C' <: C$ and $\Gamma \vdash X <: \forall(x :_D T)U$, or $P' = C'\,\forall(x :_D T')U'$, such that $\Gamma \vdash C' <: C$, $\Gamma \vdash T <: T'$, and $\Gamma, x :_D T' \vdash U' <: U$. In the first case, we invoke Lemma A.13 to show that $P = Y \wedge C_1$, $\Gamma \vdash C_1 <: C'$, and $\Gamma \vdash Y <: X$. Now we can conclude by the transitivity of subcapturing and subtyping. In the other case, we invoke IH again on the first subtype derivation and conclude by the transitivity of subcapturing and subtyping.

Case (CAPT). By IH. □

Lemma A.17 (Subtype inversion: type abstraction). *If $\Gamma \vdash P <: \forall[X <: S]T \wedge C$, then either (i) P is of the form $Y \wedge C'$, $\Gamma \vdash C' <: C$ and $\Gamma \vdash Y <: \forall[X <: S]T$, or (ii) P is of the form $\forall[X <: S']T' \wedge C'$ such that $\Gamma \vdash C' <: C$, $\Gamma \vdash S <: S'$, and $\Gamma, X <: S' \vdash T' <: T$.*

Proof. Analogous to the proof of Lemma A.16. □

Lemma A.18 (Subtype inversion: boxed term). *If $\Gamma \vdash P <: (\square\, T) \wedge C$, then either (i) P is of the form $Y \wedge C'$, $\Gamma \vdash C' <: C$ and $\Gamma \vdash Y <: \square\, T$, or (ii) P is of the form $(\square\, T') \wedge C'$ such that $\Gamma \vdash C' <: C$, and $\Gamma \vdash T' <: T$.*

Proof. Analogous to the proof of Lemma A.16. □

A.2.3 *Properties of Separation Checking.*

Lemma A.19 (Separation checking inversion: elements). *If $\Gamma \vdash C_1 \bowtie C_2$, then (1) $\forall x \in C_1$ we have $\Gamma \vdash x \bowtie C_2$; and (2) $\forall x \in C_2$ we have $\Gamma \vdash C_1 \bowtie x$.*

Proof. By induction on the derivation.

Case (NI-SYMM). Concluding by swapping the two conclusions in the IH.

Case (NI-SET). Then we have $\overline{\Gamma \vdash x \bowtie C_2}^{\,x \in C_1}$. The first part of the goal is immediate. Now we show the second part of the goal. By applying IH repeatedly we can deduce that $\forall x_1 \in C_1, \forall x_2 \in C_2$ we have $\Gamma \vdash x_1 \bowtie x_2$. We therefore show that $\forall x_2 \in C_2$ we have $\Gamma \vdash C_1 \bowtie x_2$ by (NI-SYMM) and (NI-SET), thus concluding this case.

Case (NI-DEGREE), (NI-VAR) and (NI-READER). These cases are immediate since both C_1 and C_2 are singletons. □

Lemma A.20 (Reader capability specialization). *Given any Γ and h, we have: (1) $|\Gamma \vdash \{\mathbf{rdr}\} \bowtie C_2| \le h$ implies $\Gamma \vdash \{x\} \bowtie C_2$ for every x such that is-reader$_\Gamma$ x; and (2) $|\Gamma \vdash C_1 \bowtie \{\mathbf{rdr}\}| \le h$ implies $\Gamma \vdash C_1 \bowtie \{x\}$ for every x such that is-reader$_\Gamma$ x.*

Proof. By induction on the derivation depth h. We prove each of the conclusion respectively, starting by analyzing the cases of the first one.

Case (NI-SYMM) and (NI-SET). By applying the IH.

Case (NI-DEGREE) and (NI-VAR). Not applicable.

Case (NI-READER). Then $C_2 = \{y\}$ and $\Gamma \vdash \{y\} <: \{\mathbf{rdr}\}$. By the (SC-READER) we can show that $\Gamma \vdash \{x\} <: \{\mathbf{rdr}\}$. This case can therefore be concluded using the (NI-READER) rule.

Now we inspect the derivation in the second case.

Case (NI-SYMM). By the IH.

Case (NI-SET). Then $\overline{\Gamma \vdash x \bowtie \{\mathbf{rdr}\}}^{\,x \in C_1}$. By the IH we can show that given any y such that is-reader$_\Gamma$ y, we can show that $\overline{\Gamma \vdash x \bowtie y}^{\,x \in C_1}$. We can therefore conclude this case by the (NI-SET) rule.

Case (NI-DEGREE). Not applicable.

Case (NI-VAR). By the IH and the same rule.

Case (NI-READER). This case can be concluded analogously to the one in the previous subgoal. □

Lemma A.21 (Universal capability specialization). *Given any* Γ *and* h*, if (1)* $|\Gamma \vdash \{\mathbf{cap}\} \bowtie C_2| \leq h$ *implies* $\Gamma \vdash C \bowtie C_2$ *for any* C*, and (2)* $|\Gamma \vdash C_1 \bowtie \{\mathbf{cap}\}| \leq h$*; then* $\Gamma \vdash C_1 \bowtie C$ *for any* C*.*

Proof. By induction on the derivation depth h. We establish the two conclusions respectively. We start by showing that we can prove the first one in each case.

Case (NI-SYMM) and (NI-SET). By using the IH.

Case (NI-DEGREE) and (NI-VAR). Not applicable since $\mathbf{cap} \notin \mathrm{dom}(\Gamma)$.

Case (NI-READER). Then $\Gamma \vdash \{\mathbf{cap}\} <: \{\mathbf{rdr}\}$. By induction on the subcapturing derivation we can derive a contradiction in each case, rendering this case impossible.

Then we show that we can prove the second conclusion in each case. *Case (NI-SYMM) and (NI-SET).* By the IH.

Case (NI-DEGREE). Then $C_1 = \{x\}, x :_D T \in \Gamma$ and $\mathbf{cap} \in D$. This is contradictory since the D cannot contain $\mathbf{cap}$.

Case (NI-VAR). Then $C_1 = \{x\}, x : S \wedge C' \in \Gamma$ and $\Gamma \vdash \{\mathbf{cap}\} \bowtie C'$. By the IH we can show that $\Gamma \vdash C \bowtie C'$ for any C, and conclude by the (NI-VAR) rule.

Case (NI-READER). Similarly we have $\Gamma \vdash \{\mathbf{cap}\} <: \{\mathbf{rdr}\}$ and derive a contradiction from it. □

Lemma A.22 (Subcapture preserves separation). *If* $\Gamma \vdash C_1 \bowtie C_2$ *and* $\Gamma \vdash C_0 <: C_1$*, then* $\Gamma \vdash C_0 \bowtie C_2$.

Proof. By induction on the subcapture derivation.

Case (SC-TRANS). Then $\Gamma \vdash C_0 <: C$ and $\Gamma \vdash C <: C_1$ for some C. We conclude by applying the IH twice.

Case (SC-VAR). Then $C_0 = \{x\}, x : S \wedge C \in \Gamma$, and $\Gamma \vdash C <: C_1$. By IH we can show that $\Gamma \vdash C \bowtie C_2$, and conclude by the (NI-VAR) rule.

Case (SC-ELEM). Then $C_0 = \{x\}$ and $x \in C_1$. We conclude this case by Lemma A.19.

Case (SC-SET). Then $\overline{\Gamma \vdash x \bowtie C_2}^{x \in C_1}$. This case can be concluded by applying IH repeatedly and using the (NI-SET) rule.

Case (SC-READER). Then $C_0 = \{x\}$, $\text{is-reader}_\Gamma\ x$, and $C_1 = \{\mathbf{rdr}\}$. Now we can conclude this case by invoking Lemma A.20.

Case (SC-RDR-CAP). Then $C_0 = \{\mathbf{rdr}\}$ and $C_1 = \{\mathbf{cap}\}$. Now we conclude by invoking Lemma A.21.

Case (SC-REF-CAP). Analogous to the case for (SC-RDR-CAP). □

Corollary A.23 (Set inclusion preserves separation). *If* $\Gamma \vdash C_1 \bowtie C_2$ *and* $C_0 \subseteq C_1$*, then* $\Gamma \vdash C_0 \bowtie C_2$.

Lemma A.24 (Evaluation context reification over subcapture). *If (i)* $\Gamma \vdash e \sim \Delta$*, (ii)* $\Gamma, \Delta \vdash cv(s') <: cv(s)$*, and (iii)* s *is not a value, then* $\Gamma \vdash cv(e\,[\,s'\,]) <: cv(e\,[\,s\,])$.

Proof. By induction on e.

Case $e = []$. Immediate.

Case $e = let_m\ x = e'\ in\ u$. Then we have $\mathrm{cv}(e\,[\,s\,]) = \mathrm{cv}(e'\,[\,s\,]) \cup \mathrm{cv}(u) \setminus \{x\}$, and $\mathrm{cv}(e\,[\,s'\,]) \subseteq \mathrm{cv}(e'\,[\,s'\,]) \cup \mathrm{cv}(u) \setminus \{x\}$. By the reflexivity of subcapturing we have $\Gamma \vdash \mathrm{cv}(u) \setminus \{x\} <: \mathrm{cv}(u) \setminus \{x\}$. By IH, we have $\Gamma \vdash \mathrm{cv}(e'\,[\,s'\,]) <: \mathrm{cv}(e'\,[\,s'\,])$. By Corollary A.6, we have $\Gamma \vdash \mathrm{cv}(e'\,[\,s'\,]) \cup \mathrm{cv}(u) \setminus \{x\} <: \mathrm{cv}(e'\,[\,s'\,]) \cup \mathrm{cv}(u) \setminus \{x\}$. We can therefore conclude this case.

Case $e = let_{par}\ x = t\ in\ e'$. By inspecting the derivation of $\Gamma \vdash e \sim \Delta$, we can show that $\Delta = x :_D T, \Delta'$, $\Gamma \vdash t : T$, and $\Gamma, x :_D T \vdash e' \sim \Delta'$. Also, we have $\mathrm{cv}(e\,[\,s\,]) = \mathrm{cv}(t) \cup \mathrm{cv}(e'\,[\,s\,]) \setminus \{x\}$ and $\mathrm{cv}(e\,[\,s'\,]) = \mathrm{cv}(t) \cup \mathrm{cv}(e'\,[\,s'\,]) \setminus \{x\}$. By the reflexivity of subcapture, we have $\Gamma \vdash \mathrm{cv}(t) <: \mathrm{cv}(t)$. By IH, we have $\Gamma, x :_D T \vdash \mathrm{cv}(e'\,[\,s'\,]) <: \mathrm{cv}(e'\,[\,s\,])$. Now we invoke Lemma A.12 and show that $\Gamma \vdash \mathrm{cv}(e'\,[\,s'\,]) \setminus \{x\} <: \mathrm{cv}(e'\,[\,s\,]) \setminus \{x\}$. We can conclude this case by Corollary A.6. □

Lemma A.25 (Subcapture to sub-interference). *Given the environment* Γ, Δ *and two terms* s, s'*, if (i)* s *is not a value, and (ii)* $\Gamma, \Delta \vdash cv(s') <: cv(s)$*, then* $\Gamma; \Delta \vdash s' \triangleleft: s$.

Proof. Given any Δ_1, Δ_2 and e such that $\Delta = \Delta_1, \Delta_2$ and $\Gamma, \Delta_1 \vdash e \sim \Delta_2$, the goal is to show that $\forall C\ \Gamma, \Delta_1 \vdash \mathrm{cv}(e[s]) \bowtie C$ implies $\Gamma, \Delta_1 \vdash \mathrm{cv}(e[s']) \bowtie C$. By Lemma A.24 we have $\Gamma, \Delta_1 \vdash \mathrm{cv}(e[s']) <: \mathrm{cv}(e[s])$. Now we conclude this case by Lemma A.22. □

Corollary A.26 (Captured set inclusion implies sub-interference). *If* $cv(s') \subseteq cv(s)$ *and* s *is not a value, then* $\Gamma; \Delta \vdash s' \triangleleft: s$.

Lemma A.27 (Widening preserves separation checking). *Given an inert environment* Γ*, a variable* $x :_D T \in \Gamma$ *where* $\mathbf{ref} \notin cs(T)$ *and* $\Gamma \vdash \{x\} <: \{\mathbf{rdr}\}$ *does not hold, and a natural number* h*, then (1)* $|\Gamma \vdash \{x\} \bowtie C| \leq h$ *implies* $\Gamma \vdash cs(T) \bowtie C$*, and (2)* $|\Gamma \vdash C \bowtie \{x\}| \leq h$ *implies* $\Gamma \vdash C \bowtie cs(T)$*. Here,* $|\Gamma \vdash C_1 \bowtie C_2|$ *denotes the height of the derivation tree.*

Proof. By induction on the lexical order $(\llbracket C \rrbracket, h)$. We prove the two conclusions respecitively. We start by establishing the first one, by a case analysis on the last rule applied in the derivation of $\Gamma \vdash \{x\} \bowtie C$.

Case (NI-SYMM). Then $\Gamma \vdash C \bowtie \{x\}$. This case is concluded by the IH and the same rule.

Case (NI-SET). By the IH.

Case (NI-DEGREE). Then $C = \{y\}$ and $y \in D$. By the inertness of Γ and that $\mathbf{ref} \notin \mathrm{cs}(T)$, we have $\Gamma \vdash D \bowtie \mathrm{cs}(T)$. By Lemma A.19, $\Gamma \vdash y \bowtie \mathrm{cs}(T)$. Now we conclude by (NI-SYMM).

Case (NI-VAR). Then $\Gamma \vdash \mathrm{cs}(T) \bowtie y$ and we can conclude this case immediately.

Case (NI-READER). This case is not applicable.

The first conclusion is therefore proven. Now we proceed to the second one.

Case (NI-SYMM). By IH.

Case (NI-SET). Then $C = \{y_i\}_{i=1,\cdots,n}$, and $\Gamma \vdash y_i \bowtie x$ for $i = 1, \cdots, n$. By repeated IH, we can demonstrate that $\overline{\Gamma \vdash y_i \bowtie C}^{i=1,\cdots,n}$. Now we conclude this case by (NI-SET).

Case (NI-DEGREE). Then $C = \{x'\}$, $x' :_{D'} T' \in \Gamma$, and $x \in D'$. If $\mathbf{ref} \in \mathrm{cs}(T')$, then by the inertness of Γ, we have $D' = \mathrm{dom}(\Gamma_0)$ if we decompose the environment into $\Gamma = \Gamma_0, x' :_{D'} T', \Gamma_1$. Since $x \in D'$, by the well-formedness of the environment, x is bound in Γ_0. Again by the well-formedness, we have $\mathrm{cs}(T) \subseteq \mathrm{dom}(\Gamma_0) = D'$. We can therefore prove the goal by repeated (NI-DEGREE) and (NI-SET). Otherwise, if $\mathbf{ref} \notin \mathrm{cs}(T')$, we have $\Gamma \vdash D' \bowtie \mathrm{cs}(T')$. By Lemma A.19 we have $\Gamma \vdash \{x\} \bowtie \mathrm{cs}(T')$. Note that $\llbracket \mathrm{cs}(T') \rrbracket < \mathrm{depth}_\Gamma(x')$ by the well-formedness of Γ. We can therefore invoke IH and show that $\Gamma \vdash \mathrm{cs}(T) \bowtie \mathrm{cs}(T')$ and conclude the case by (NI-SYMM).

Case (NI-VAR). Then $C = \{x'\}$, $x' :_{D'} T' \in \Gamma$, and $\Gamma \vdash \mathrm{cs}(T') \bowtie x$. We conclude this case by IH and the same rule.

Case (NI-READER). Not applicable. □

Lemma A.28 (Widening implies sub-interference). *Given* Γ, Δ, s, s' *where* s *is not a value, if (i)* Γ, Δ *is inert, (ii)* $x :_D T \in \Gamma$ *where* $x \in cv(s)$ *and* $\mathbf{ref} \notin cs(T)$*, (iii)* $cv(s') \subseteq cv(s) - \{x, x'\} \cup cs(T)$*, then* $\Gamma; \Delta \vdash s' \triangleleft: s$.

Proof. Given Δ_1, Δ_2, e such that $\Delta = \Delta_1, \Delta_2$ and $\Gamma, \Delta_1 \vdash e \sim \Delta_2$, and we have $\Gamma \vdash \mathrm{cv}(e[s]) \bowtie C$. Since s is not a value, we have $x \in \mathrm{cv}(s) \subseteq \mathrm{cv}(e[s])$. By Lemma A.19 we show that $\Gamma \vdash x \bowtie C$. By Lemma A.27, we have $\Gamma \vdash \mathrm{cs}(T) \bowtie C$. Therefore, $\Gamma \vdash \mathrm{cv}(e[s]) \cup \mathrm{cs}(T) \bowtie C$. Now we show that $\mathrm{cv}(e[s']) \subseteq \mathrm{cv}(e[s]) \cup \mathrm{cs}(T)$ and conclude this case. □

Lemma A.29 (Shifting boundary of sub-interference). *Given* $\Gamma, \Delta_1, \Delta_2, \Gamma; \Delta_1, \Delta_2 \vdash s' \triangleleft: s$ *implies* $\Gamma, \Delta_1; \Delta_2 \vdash s' \triangleleft: s$.

Proof. Given any Δ_3, Δ_4, e such that $\Delta_2 = \Delta_3, \Delta_4$ and $\Gamma, \Delta_1, \Delta_3 \vdash e \sim \Delta_4$, if $\Gamma, \Delta_1, \Delta_3 \vdash e\,[\,s\,] \bowtie C$, we can show that $\Gamma, \Delta_1, \Delta_3 \vdash e\,[\,s'\,] \bowtie C$ by the premise. □

Lemma A.30 (Subtyping preserves is-reader). *If is-reader$_\Gamma$ T and $\Gamma \vdash T' <: T$, then is-reader$_\Gamma$ T'.*

Proof. By induction on the derivation of $\Gamma \vdash T' <: T$. Note that by inspecting the derivation of is-reader$_\Gamma$ T we know that T is either a type variable or a reader, which implies that only the following cases are applicable.

Case (refl). Then $T' = T$, and we conclude immediately from the premise.

Case (tvar). Then $T' = X$, $T = S$, and $X <: S \in \Gamma$. We conclude immediately by the (rd-tvar) rule. □

A.2.4 Structural Properties of Typing.

Lemma A.31 (Permutation). *Given Γ, and Δ which is a well-formed environment permuted from Γ:*

(i) $\Gamma \vdash T$ ***wf*** *implies* $\Delta \vdash T$ ***wf****;*
(ii) $\Gamma \vdash D$ ***wf*** *implies* $\Delta \vdash D$ ***wf****;*
(iii) $\Gamma \vdash t : T$ *implies* $\Delta \vdash t : T$*;*
(iv) $\Gamma \vdash T <: U$ *implies* $\Delta \vdash T <: U$*;*
(v) $\Gamma \vdash C_1 <: C_2$ *implies* $\Delta \vdash C_1 <: C_2$*;*
(vi) *is-reader$_\Gamma$ x implies is-reader$_\Gamma$ x;*
(vii) $\Gamma \vdash C_1 \bowtie C_2$ *implies* $\Delta \vdash C_1 \bowtie C_2$*.*

Proof. By straightforward induction on the derivations. No rule depends on the order of the bindings. □

Lemma A.32 (Weakening). *Given Γ, Δ,*

(i) $\Gamma \vdash T$ ***wf*** *implies* $\Gamma, \Delta \vdash T$ ***wf****;*
(ii) $\Gamma \vdash D$ ***wf*** *implies* $\Gamma, \Delta \vdash D$ ***wf****;*
(iii) $\Gamma \vdash t : T$ *implies* $\Gamma, \Delta \vdash t : T$*;*
(iv) $\Gamma \vdash T <: U$ *implies* $\Gamma, \Delta \vdash T <: U$*;*
(v) $\Gamma \vdash C_1 <: C_2$ *implies* $\Gamma, \Delta \vdash C_1 <: C_2$*;*
(vi) *is-reader$_\Gamma$ x implies is-reader$_{\Gamma,\Delta}$ x;*
(vii) $\Gamma \vdash C_1 \bowtie C_2$ *implies* $\Gamma, \Delta \vdash C_1 \bowtie C_2$*.*

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). In this case, $t = \lambda(z :_D U)s$, $T = \forall(z :_D U)T'$, and $\Gamma, z :_D U \vdash s : T'$. By IH we have $\Gamma, z :_D U, \Delta \vdash s : T'$. Since D and U cannot mention variables in $\Delta, \Gamma, \Delta, z :_D U$ is still well-formed. By permutation we have $\Gamma, \Delta, z :_D U \vdash s : T'$. This case is therefore concluded by (abs). □

Lemma A.33 (Bound narrowing). *Given an environment $\Gamma, X <: S, \Delta$ and the fact that $\Gamma \vdash S' <: S$, the followings hold:*

(1) $\Gamma, X <: S, \Delta \vdash T$ ***wf*** *implies* $\Gamma, X <: S', \Delta \vdash T$ ***wf****;*
(2) $\Gamma, X <: S, \Delta \vdash D$ ***wf*** *implies* $\Gamma, X <: S', \Delta \vdash D$ ***wf****;*
(3) $\Gamma, X <: S, \Delta \vdash C_1 <: C_2$ *implies* $\Gamma, X <: S', \Delta \vdash C_1 <: C_2$*;*
(4) $\Gamma, X <: S, \Delta \vdash T <: U$ *implies* $\Gamma, X <: S', \Delta \vdash T <: U$*;*
(5) *is-reader$_{\Gamma, X<:S, \Delta}$ x implies is-reader$_{\Gamma, X<:S', \Delta}$ x;*
(6) $\Gamma, X <: S, \Delta \vdash C_1 \bowtie C_2$ *implies* $\Gamma, X <: S', \Delta \vdash C_1 \bowtie C_2$*;*

(7) $\Gamma, X <: S, \Delta \vdash t : T$ *implies* $\Gamma, X <: S', \Delta \vdash t : T$.

Proof. By straightforward induction on the derivations. The only two rules that make use of type variable bounds are (TVAR) in subtyping, and (RD-TVAR) in is-reader. In all other cases, we can conclude it by utilizing the IH, other narrowing lemmas and the same rule. We will now give the proof of the (TVAR) case.

Case (TVAR). In this case, $T_1 = Y$, $T_2 = R$, and $Y <: R \in \Gamma, X <: S, \Delta$. If $Y \neq X$, then $Y <: R \in \Gamma, X <: S', \Delta$. Applying the (TVAR) rule allows us to derive the conclusion. Otherwise, $X = Y$. The goal becomes $\Gamma, X <: S', \Delta \vdash X <: S$. By weakening we can show that $\Gamma, X <: S', \Delta \vdash S' <: S$. We can therefore conclude this case by (TVAR) and (TRANS). □

Lemma A.34 (Type narrowing). *Given* $\Gamma, x : P, \Delta$, *and that* $\Gamma \vdash P' <: P$, *the following propositions hold:*

(1) $\Gamma, x : P, \Delta \vdash T$ ***wf*** *implies* $\Gamma, x : P', \Delta \vdash T$ ***wf****;*
(2) $\Gamma, x : P, \Delta \vdash D$ ***wf*** *implies* $\Gamma, x : P', \Delta \vdash D$ ***wf****;*
(3) $\Gamma, x : P, \Delta \vdash C_1 <: C_2$ *implies* $\Gamma, x : P', \Delta \vdash C_1 <: C_2$*;*
(4) $\Gamma, x : P, \Delta \vdash T_1 <: T_2$ *implies* $\Gamma, x : P', \Delta \vdash T_1 <: T_2$*;*
(5) *is-reader*$_{\Gamma, x:P, \Delta}\ y$ *implies is-reader*$_{\Gamma, x:P', \Delta}\ y$*;*
(6) $\Gamma, x : P, \Delta \vdash C_1 \bowtie C_2$ *implies* $\Gamma, x : P', \Delta \vdash C_1 \bowtie C_2$*;*
(7) $\Gamma, x : P, \Delta \vdash t : T$ *implies* $\Gamma, x : P', \Delta \vdash t <: T$.

Proof. The proof follows through a straightforward induction on the derivations, wherein only the cases enumerated below are contigent to the variable bindings in the context. The other cases can be deduced by employing the IH, the same rule and other narrowing lemmas.

Case (VAR). In this case, $t = y$, $y : S \wedge C \in \Gamma, x : P, \Delta$, and $T = \{x\}\ S$. If $x \neq y$, then the binding for y is remains unaffected and we can directly conclude by applying the (VAR) rule. Otherwise, we have $x = y$, implying that $P = S \wedge C$. We can demonstrate that $P' = S' \wedge C'$ for some C', S'. Inspecting the judgment $\Gamma \vdash P' <: P$ we can show that $\Gamma \vdash S' <: S$. Consequently, We can establish that $\Gamma, x : S', \Delta \vdash \{x\}\ S' <: \{x\}\ S$ by applying the weakening lemma and the (CAPT) rule. Finally, we can conclude by utilizing the (VAR) and the (SUB) rules.

Case (SC-VAR). In this case, $C_1 = \{y\}$, $y : S \wedge C \in \Gamma, x : P, \Delta$, and $\Gamma, x : P, \Delta \vdash C <: C_2$. Using the IH, we can demonstrate that $\Gamma, x : P', \Delta \vdash C <: C_2$. If $x \neq y$, the binding for y remains unaffected and we can immediately apply the IH and the (SC-VAR) rule to conclude. Otherwise, we have $x = y$, implying that $P = S \wedge C$. Inpsecting the subtype judgment $\Gamma \vdash P' <: P$, we can show that $\Gamma \vdash C' <: C$ where C' denotes cs (P'). Invoking the weakening lemma and the subcapturing transitivity lemma, we can derive that $\Gamma, x : P', \Delta \vdash C' <: C_2$. Finally, we conclude by applying the (SC-VAR) rule.

Case (NI-VAR). In this case, $C_1 = \{y\}$, $C_2 = \{z\}\ y : S \wedge C \in \Gamma, x : P, \Delta$ and $\Gamma, x : P, \Delta \vdash C \bowtie z$. Using IH, we can demonstrate that $\Gamma, x : P', \Delta \vdash C \bowtie z$. If $y \neq x$, then the binding for y stays unaffected and we can immediately invoke the IH and the (NI-VAR) rule to conclude. Otherwise we have $y = x$, implying that $P = S \wedge C$. From the subtype judgment $\Gamma \vdash P' <: P$ we can show that $\Gamma \vdash C' <: C$ where C' is cs (P'). We invoke the weakning lemma and Lemma A.22 to prove that $\Gamma, x : P', \Delta \vdash C' \bowtie z$. Finally, we can apply the (NI-VAR) rule to conclude this case. □

Lemma A.35 (Separation degree expansion). *Given* $\Gamma, x :_D T, \Delta$, *and that* $D \subseteq D'$, *the following propositions hold:*

- $\Gamma, x :_D T, \Delta \vdash T$ ***wf*** *implies* $\Gamma, x :_{D'} T, \Delta \vdash T$ ***wf****;*
- $\Gamma, x :_D T, \Delta \vdash D$ ***wf*** *implies* $\Gamma, x :_{D'} T, \Delta \vdash D$ ***wf****;*
- $\Gamma, x :_D T, \Delta \vdash C_1 <: C_2$ *implies* $\Gamma, x :_{D'} T, \Delta \vdash C_1 <: C_2$*;*
- $\Gamma, x :_D T, \Delta \vdash T_1 <: T_2$ *implies* $\Gamma, x :_{D'} T, \Delta \vdash T_1 <: T_2$*;*

- *is-reader*$_{\Gamma,x:_D T,\Delta}\ y$ *implies is-reader*$_{\Gamma,x:_{D'} T,\Delta}\ y$;
- $\Gamma, x :_D T, \Delta \vdash C_1 \bowtie C_2$ *implies* $\Gamma, x :_{D'} T, \Delta \vdash C_1 \bowtie C_2$;
- $\Gamma, x :_D T, \Delta \vdash t : U$ *implies* $\Gamma, x :_{D'} T, \Delta \vdash t : U$.

Proof. The proof is carried out through straightforward induction on the derivations. Only the (NI-DEGREE) case depends on the separation degrees. Other cases can be concluded by IH, other expansion lemmas and the same rule.

Case (NI-DEGREE). Then $C_1 = \{z_1\}$, $C_2 = \{z_2\}$, $z_1 :_{D_1} T_1 \in \Gamma, x :_D T, \Delta$, and $z_2 \in D_1$. If $z_1 = x$, then $D_1 = D$. Considering $D \subseteq D'$, we can show that $z_2 \in D'$. Finally, we invoke (NI-VAR) and conclude. □

A.2.5 Properties of Evaluation Configurations.

Lemma A.36 (Value typing (I)). *If* $\Gamma \vdash v : T$ *then* T *is not in the form of* $X \wedge C$.

Proof. By straightforward induction on the derivation.

Case (SUB). Then $\Gamma \vdash v : T'$, and $\Gamma \vdash T' <: T$. By IH we know that T' is not in the form of $X \wedge C$. If T is in the form of $X \wedge C$, then we invoke Lemma A.13 and derive a contradiction. We therefore conclude.

Other cases. Other cases for typing values are immediate. □

Lemma A.37 (Value typing (II)). *If* $\Gamma \vdash v : T$ *then* T *is not in the form of* $\mathsf{Ref}[S] \wedge C$.

Proof. By straightforward induction on the typing derivation. No typing rule for values results in a $\mathsf{Ref}[S] \wedge C$ type. In the (SUB) case, we have $\Gamma \vdash v : T'$ and $\Gamma \vdash T' <: T$. By IH we have T' is not in the form of $\mathsf{Ref}[S] \wedge C$. By induction on the $\Gamma \vdash T' <: T$ we can show that T is also not in this form. We can therefore conclude this case. □

Lemma A.38 (Store lookup: mutable variables). *If (i)* $\vdash \gamma \sim \Gamma$, *(ii)* $\Gamma \vdash x : \mathit{Ref}[S] \wedge C$, *then* $\exists v.\gamma(\mathit{var}\ x) = v$.

Proof. By induction on $\vdash \gamma \sim \Gamma$.

Case (ST-EMPTY). Then $\gamma = \Gamma = \varnothing$. It is contradictory to have $\varnothing \vdash x : \mathsf{Ref}[S] \wedge C$.

Case (ST-VAL). Then $\gamma = \gamma_0, \mathsf{val}_D\ y \mapsto v$ $\Gamma = \Gamma_0, y :_D \mathsf{cv}(v)\ S'$, and $\Gamma_0 \vdash v : \mathsf{cv}(v)\ S'$.

- If $x = y$, by induction on the typing judgment and applying weakening, we can show that $\Gamma \vdash S' \wedge \mathsf{cv}(v) <: \mathsf{Ref}[S] \wedge C$. By Lemma A.14 and Lemma A.36, we can show that $\Gamma \vdash \mathsf{cv}(v) <: C$ and $S' = \mathsf{Ref}[S]$. By Lemma A.37 we can derive the contradiction.
- If $x \neq y$, then we can show that $x \in \mathsf{dom}(\Gamma_0)$, and therefore $\Gamma_0 \vdash x : C\ \mathsf{Ref}[S]$. We conclude this case by IH.

Case (ST-VAR). Then $\gamma = \gamma_0, \mathsf{var}\ y = v$, and $\Gamma = \Gamma_0, y :_{\mathsf{dom}(\Gamma_0)} \mathsf{Ref}[S'] \wedge \{\mathbf{ref}\}$. If $y = x$, we conclude this case by the fact that $\gamma(\mathsf{var}\ x) = v$. Otherwise, we have $\Gamma_0 \vdash x : C\ \mathsf{Ref}[S]$ and conclude again by IH.

Case (ST-SET). Analogous to the previous case. □

Lemma A.39 (Store lookup: pure values). *If (i)* $\vdash \gamma \sim \Gamma$, *(ii)* $\Gamma \vdash x : S$, *then* $\exists v.\gamma(\mathit{val}\ x) = v$.

Proof. By induction on the derivation of $\vdash \gamma \sim \Gamma$.

Case (ST-EMPTY). Contradictory.

Case (ST-VAL). Then $\gamma = \gamma_0, \mathsf{val}\ y \mapsto v$, $\vdash \gamma_0 \sim \Gamma_0, \Gamma_0 \vdash v : T$, and $\Gamma = \Gamma_0, y :_{\{\}} T$. If $x \neq y$ we conclude by IH. Otherwise we have v such that $\gamma(\mathsf{val}\ x) = v$.

Case (ST-VAR). Then $\gamma = \gamma_0, \mathsf{var}\ y = v$, $\vdash \gamma_0 \sim \Gamma_0, \Gamma_0 \vdash S' :$, and $\Gamma = \Gamma_0, y :_{\mathsf{dom}(\Gamma_0)} \mathsf{Ref}[S'] \wedge \{\mathbf{ref}\}$. If $x = y$, we can show that $\Gamma \vdash \{\mathbf{ref}\} <: \{\}$ which is contradictory. Otherwise we conclude the case by IH.

Case (ST-SET). By IH. □

Lemma A.40 (Store lookup: term abstractions). *If (i)* $\vdash \gamma \sim \Gamma$, *(ii)* $\Gamma \vdash x : \forall(z :_D U)T \,\hat{}\, C$, *then* $\exists v.\gamma(\mathit{val}\, x) = v$.

Proof. By induction on the $\vdash \gamma \sim \Gamma$ derivation.
Case (ST-EMPTY). Contradictory.
Case (ST-VAL). Then $\gamma = \gamma_0, \mathsf{val}\, y \mapsto v$, $\vdash \gamma_0 \sim \Gamma_0, \Gamma_0 \vdash v : T$, and $\Gamma = \Gamma_0, y :_D T$. If $x = y$, then we conclude immediately. Otherwise we conclude the goal by IH.
Case (ST-VAR). Then $\gamma = \gamma_0, \mathsf{var}\, y = v$, $\vdash \gamma_0 \sim \Gamma_0, \Gamma_0 \vdash v : S'$, and $\Gamma = \Gamma_0, y :_{\mathsf{dom}(\Gamma_0)} \mathsf{Ref}[S'] \,\hat{}\, \{\mathbf{ref}\}$. If $x = y$, we can show that $\Gamma \vdash \mathsf{Ref}[S'] \,\hat{}\, \{\mathbf{ref}\} <: \forall(z :_D T)U \,\hat{}\, C$. By Lemma A.16, we can derive a contradiction. Otherwise, $x \neq y$. We conclude this case by IH.
Case (ST-SET). As above. □

Lemma A.41 (Store lookup: type abstractions). *If (i)* $\vdash \gamma \sim \Gamma$, *(ii)* $\Gamma \vdash x : \forall[X <: S]T \,\hat{}\, C$, *then* $\exists v.\gamma(\mathit{val}\, x) = v$.

Proof. Analogous to the proof of Lemma A.40. □

Lemma A.42 (Store lookup: reader). *If (i)* $\vdash \gamma \sim \Gamma$, *(ii)* $\Gamma \vdash x : \mathit{Rdr}[S] \,\hat{}\, C$, *then* $\exists v.\gamma(\mathit{val}\, x) = v$.

Proof. Analogous to the proof of Lemma A.40. □

Lemma A.43 (Inversion of evaluation contexts). *If* $\Gamma \vdash e\,[\,s\,] : T$, *then* $\exists \Delta, U$ *such that (i)* $\Gamma \vdash e \sim \Delta : [U \mathbin{@} s] \Rightarrow T$, *and (ii)* $\Gamma, \Delta \vdash s : U$.

Proof. By induction on e.
Case $e = []$. Set $\Delta = \emptyset$ and $U = T$. This case can be concluded immediately.
Case $e = \mathit{let}_m\, x = e'\ \mathit{in}\, u$. Then $\Gamma \vdash \mathsf{let}_m\, x = e'\,[\,s\,]\ \mathsf{in}\, u : T$. By induction on this typing derivation. The sub-goal is that $\Gamma \vdash e'\,[\,s\,] : T_0$ for some T_0, $\Gamma, x :_{\{\}} T_0 \vdash u : T$, $x \notin \mathsf{fv}(T)$, and $\Gamma \vdash e'\,[\,s\,] \bowtie u$ if $m = \mathsf{par}$.

- *Case (LET).* Conclude immediately from the premises.
- *Case (SUB).* In this case, $\Gamma \vdash \mathsf{let}_{\mathsf{par}}\, x = e'\,[\,s\,]\ \mathsf{in}\, u : T'$ for some T', and $\Gamma \vdash T' <: T$. We conclude this case by IH and (SUB).
- *Other cases.* Not applicable.

Now we invoke IH on $\Gamma \vdash e'\,[\,s\,] : T_0$ and show that $\exists \Delta_0, U_0$ such that $\Gamma \vdash e' \sim \Delta_0 : [U_0 \mathbin{@} s] \Rightarrow T_0$. Now we set $\Delta = \Delta_0$ and $U = U_0$. First, we have $\Gamma \vdash e \sim \Delta$ by (EV-LET-1). Given s' such that $\Gamma, \Delta \vdash s' : U$ and $\Gamma; \Delta \vdash s' \triangleleft: s$, we have $\Gamma \vdash e'\,[\,s'\,] : T_0$ by IH. Also, since we have $\Gamma \vdash e' \sim \Delta$, from $\Gamma; \Delta \vdash s' \triangleleft: s$ we can show that $\Gamma \vdash e'\,[\,s'\,] \bowtie u$. We can conclude this case by (LET).

Case $e = \mathit{let}_{\mathit{par}}\, x = t\ \mathit{in}\, e'$. Then $\Gamma \vdash \mathsf{let}_{\mathsf{par}}\, x = t\ \mathsf{in}\, e'\,[\,s\,] : T$. By a similar induction on this typing derivation, we can show that $\Gamma \vdash t : T_0$ for some T_0, $\Gamma, x :_{\{\}} T_0 \vdash e'\,[\,s\,] : T$, $x \notin \mathsf{fv}(T)$, and $\Gamma \vdash t \bowtie e'\,[\,s\,]$. Now we invoke IH to show that $\exists \Delta_0, U_0$ such that $\Gamma, x :_{\{\}} T_0 \vdash e' \sim \Delta_0 : [U_0 \mathbin{@} s] \Rightarrow T$, and $\Gamma, x :_{\{\}} T_0, \Delta_0 \vdash s : U_0$. Set $\Delta = x :_{\{\}} T_0, \Delta_0$ and $U = U_0$. We have $\Gamma \vdash e \sim x :_{\{\}} T_0, \Delta_0$ by (EV-LET-2). $\forall s'$ such that $\Gamma, \Delta \vdash s' : U$ and $\Gamma; \Delta \vdash s' \triangleleft: s$. the goal is to show that $\Gamma \vdash \mathsf{let}_{\mathsf{par}}\, x = t\ \mathsf{in}\, e'\,[\,s'\,] : T$. By Lemma A.29, we have $\Gamma, x :_{\{\}} T_0; \Delta_0 \vdash s' \triangleleft: s$. By IH we can show that $\Gamma, x :_{\{\}} T_0 \vdash e'\,[\,s'\,] : T$. From $\Gamma; \Delta \vdash s' \triangleleft: s$ we can show that $\Gamma, x :_{\{\}} T_0 \vdash \mathsf{cv}(t) \bowtie \mathsf{cv}(e'\,[\,s'\,])$. By strengthening, we show that $\Gamma \vdash \mathsf{cv}(t) \bowtie \mathsf{cv}(e'\,[\,s'\,]) \setminus \{x\}$, which is the same as $\Gamma \vdash t \bowtie e'\,[\,s'\,]$. This case is therefore concluded by (LET). □

Lemma A.44 (Weakening of evaluation context inversion). $\Gamma \vdash e \sim \Delta : [U \mathbin{@} s] \Rightarrow T$ *implies* $\Gamma, \Delta \vdash e \sim \Delta : [U \mathbin{@} s] \Rightarrow T$.

Proof. By weakening of environment matching, typing and separation checking. □

Lemma A.45 (Downgrading separation degree preserves subcapturing). *Given* $\Gamma = \Gamma_1, x :_{dom(\Gamma_1)} S^\wedge\{\mathbf{ref}\}, \Gamma_2$, *and* $y :_D T \in \Gamma_1$ *where* $cs(T) \cap \{\mathbf{cap}, \mathbf{rdr}, \mathbf{ref}\} = \{\}$, $\Gamma \vdash T_1 <: T_2$ *implies* $\Gamma' \vdash T_1 <: T_2$ *where* $\Gamma' = \Gamma_2, x :_{dom(\Gamma_1)/y} S^\wedge\{\mathbf{ref}\}, \Gamma_2$.

Proof. By straightfoward induction on the derivation, wherein no rule makes use of the separation degrees in the context. □

Lemma A.46 (Downgrading separation degree preserves separation checking). *Given* $\Gamma = \Gamma_1, x :_{dom(\Gamma_1)} S^\wedge\{\mathbf{ref}\}, \Gamma_2$, *and* $y :_D T \in \Gamma_1$ *where* $cs(T) \cap \{\mathbf{cap}, \mathbf{rdr}, \mathbf{ref}\} = \{\}$, $\Gamma \vdash C_1 \bowtie C_2$ *implies* $\Gamma' \vdash C_1 \bowtie C_2$ *where* $\Gamma' = \Gamma_1, x :_{dom(\Gamma_1)/y} S^\wedge\{\mathbf{ref}\}, \Gamma_2$.

Proof. By induction on the separation checking derivation.

Case (NI-SYMM). By IH and (NI-SYMM) again.

Case (NI-SET). Then $C_1 = \{x_i\}_{i=1,\cdots,n}$, and $\overline{\Gamma \vdash \{x_i\} \bowtie C_2}^{i=1,\cdots,n}$. By repeated IH, we have $\overline{\Gamma' \vdash \{x_i\} \bowtie C_2}^{i=1,\cdots,n}$. This case is therefore concluded by (NI-SET).

Case (NI-DEGREE). Then $C_1 = \{z_1\}$, $C_2 = \{z_2\}$, $z_1 :_{D_1} T_1 \in \Gamma$, and $z_2 \in D_1$. If $z_1 = x$ and $z_2 = y$, Since by the well-formedness of the environment and that $cs(T)$ does not contain root capabilities, we have $cs(T) \subseteq dom(\Gamma)$. We can therefore show that $\Gamma \vdash \{x\} \bowtie cs(T)$ by (NI-VAR) and (NI-SET). Otherwise the goal follows directly from the preconditions.

Case (NI-VAR). By applying the IH and the same rule.

Case (NI-READER). By applying Lemma A.45 and the same rule. □

Lemma A.47 (Downgrading separation degree preserves subtyping). *Given* $\Gamma = \Gamma_1, x :_{dom(\Gamma_1)} S^\wedge\{\mathbf{ref}\}, \Gamma_2$, *and* $y :_D T \in \Gamma_1$ *where* $cs(T) \cap \{\mathbf{cap}, \mathbf{rdr}, \mathbf{ref}\} = \{\}$, $\Gamma \vdash T <: U$ *implies* $\Gamma' \vdash T <: U$ *where* $\Gamma' = \Gamma_1, x :_{dom(\Gamma_1)/y} S^\wedge\{\mathbf{ref}\}, \Gamma_2$.

Proof. By straightforward induction on the subtyping derivation. No rule makes use of the separation degree on the bindings. □

Lemma A.48 (Downgrading separation degree preserves typing). *Given* $\Gamma = \Gamma_1, x :_{dom(\Gamma_1)} S^\wedge\{\mathbf{ref}\}, \Gamma_2$, *and* $y :_D T \in \Gamma_1$ *where* $cs(T) \cap \{\mathbf{cap}, \mathbf{rdr}, \mathbf{ref}\}$, $\Gamma \vdash t : T$ *implies* $\Gamma' \vdash t : T$ *where* $\Gamma' = \Gamma_1, x :_{dom(\Gamma_1)/y} S^\wedge\{\mathbf{ref}\}, \Gamma_2$.

Proof. By induction on the typing derivation.

Case (VAR). By the precondition and the same rule.

Case (SUB). By IH, Lemma A.47, and the same rule.

Case (APP) and (LET). By IH, Lemma A.46 and the same rule.

Other cases. By IH and the same rule. □

Lemma A.49 (Evaluation context inversion and reification for mutable variables). $\Gamma \vdash e\,[\,\mathit{var}\ x = y\ \mathit{in}\ s\,] : T$ *implies* $\exists \Delta, S, U$ *such that (i)* $\Gamma \vdash e \sim \Delta$, *(ii)* $\Gamma, \Delta \vdash y : S$, *(iii)* $\Gamma, x :_{dom(\Gamma)} Ref[S]^\wedge\{\mathbf{ref}\}, \Delta \vdash s : U$, *and (iv)* $\Gamma, x :_{dom(\Gamma)} Ref[S]^\wedge\{\mathbf{ref}\} \vdash e\,[\,s\,] : T$.

Proof. By induction on e.

Case $e = []$. Then $\Gamma \vdash \mathsf{var}\ x = y\ \mathsf{in}\ s : T$. By induction on the typing derivation we can show that $\Gamma \vdash y : S$ for some S, and $\Gamma, x :_D \mathrm{Ref}[S]^\wedge\{\mathbf{ref}\} \vdash s : T$ for some D. We set $\Delta = \varnothing$ and $U = T$. We can conclude this case by separation degree expansion.

Case $e = let_m\ z = e'\ in\ u$. Then $\Gamma \vdash \mathsf{let}_m\ z = e'\,[\,\mathsf{var}\ x = y\ \mathsf{in}\ s\,]\ \mathsf{in}\ u : T$. By inspecting this derivation, we can show that $\Gamma \vdash e'\,[\,\mathsf{var}\ x = y\ \mathsf{in}\ s\,] : T_0$ for some T_0, $\Gamma, z :_{\{\}} T_0 \vdash u : T$, and $\Gamma \vdash e'\,[\,\mathsf{var}\ x = y\ \mathsf{in}\ s\,] \bowtie u$. By IH, we show that $\exists \Delta_0, S_0, U_0$ such that $\Gamma \vdash e' \sim \Delta_0, \Gamma, \Delta_0 \vdash y : S_0$,

$\Gamma, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S] \wedge \{\mathbf{ref}\}, \Delta_0 \vdash s : U_0$, and $\Gamma, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S] \wedge \{\mathbf{ref}\} \vdash e\,[\,s\,] : T$. Note that $\Gamma \vdash e \sim \Delta_0$. By weakening, we can show that $\Gamma, x :_{\mathsf{dom}(\Gamma)} \vdash D' \bowtie \mathsf{cs}(T_0)$. We can show that $\mathsf{cv}(e'\,[\,s\,]) \subseteq \mathsf{cv}(e'\,[\,\mathsf{var}\ x = y\ \mathsf{in}\ s\,]) \setminus \{y\} \cup \{x\}$. Now, we can show that $\Gamma, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S] \wedge \{\mathbf{ref}\} \vdash \mathsf{cv}(e'\,[\,\mathsf{var}\ x = y\ \mathsf{in}\ s\,]) \setminus \{y\} \bowtie \mathsf{cv}(u) \setminus \{z\}$ from the precondition and weakening. Then, we can show that $\Gamma, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S] \wedge \{\mathbf{ref}\} \vdash x \bowtie \mathsf{cv}(u) \setminus \{z\}$ by (NI-DEGREE) and (NI-SET). We can therefore show that $\Gamma, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S] \wedge \{\mathbf{ref}\} \vdash e'\,[\,s\,] \bowtie u$. This case is therefore concluded by setting $\Delta = \Delta_0$ and $U = U_0$, and using the (LET) rule.

Case $e = \mathit{let}_{par}\ z = t\ \mathit{in}\ e'$. Then $\Gamma \vdash \mathsf{let}_{\mathsf{par}}\ z = t\ \mathsf{in}\ e'\,[\,\mathsf{var}\ x = y\ \mathsf{in}\ s\,] : T$. By inspecting the typing derivation we can show that $\Gamma \vdash t : T_0$ for some T_0, $\Gamma, z :_{\{\}} T_0 \vdash e'\,[\,\mathsf{var}\ x = y\ \mathsf{in}\ s\,] : T$, and $\Gamma \vdash t \bowtie e'\,[\,\mathsf{var}\ x = y\ \mathsf{in}\ s\,]$. Now we invoke IH on the typing derivation $\Gamma, z :_{\{\}} T_0 \vdash e'\,[\,\mathsf{var}\ x = y\ \mathsf{in}\ s\,] : T$ to show that $\exists \Delta_0, S_0, U_0$ such that $\Gamma, z :_{\{\}} T_0 \vdash e' \sim \Delta_0, \Gamma, z :_{\{\}} T_0, \Delta_0 \vdash y : S_0$, $\Gamma, z :_{\{\}} T_0, x :_{\mathsf{dom}(\Gamma)\cup\{z\}} \mathsf{Ref}[S_0] \wedge \{\mathbf{ref}\}, \Delta_0 \vdash s : U_0$ and $\Gamma, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S_0] \wedge \{\mathbf{ref}\}, z :_{\{\}} T_0 \vdash e'\,[\,s\,] : T$. Set $\Delta = z :_{\{\}} T_0, \Delta_0$. We can first show that $\Gamma \vdash e \sim z :_{\{\}} T_0, \Delta_0$. Also, we have $\Gamma, \Delta \vdash y : S_0$. Then we have to show that $\Gamma, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S_0] \wedge \{\mathbf{ref}\}, \Delta \vdash s : U_0$, which requires dropping z from the separation degree of x. Now we invoke Lemma A.48 to show that $\Gamma, z :_{\{\}} T_0, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S_0] \wedge \{\mathbf{ref}\}, \Delta_0 \vdash s : U_0$. Then by permutation we have $\Gamma, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S_0] \wedge \{\mathbf{ref}\}, \Delta \vdash s : U_0$. We can similarly show that $\Gamma, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S_0] \wedge \{\mathbf{ref}\}, z :_{\{\}} T_0 \vdash e'\,[\,s\,] : T$. Similar to the previous case, we can show that $\Gamma, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S] \wedge \{\mathbf{ref}\} \vdash t \bowtie e'\,[\,s\,]$. We can therefore conclude this case by (LET). □

A.2.6 Substitution.

Term Substitution.

Lemma A.50 (Term substitution preserves readers). *If (i)* $\Gamma, x : P, \Delta \vdash T \uparrow \mathit{Rdr}$, *and (ii)* $\Gamma \vdash x' : P$, *then* $\mathit{is\text{-}reader}_{\Gamma,\theta\Delta}\ \theta T$, *where* $\theta = [x \mapsto x']$.

Proof. By induction on the first derivation.

Case (RD-READER). We conclude by the same rule.

Case (RD-TVAR). Then $T = X \wedge C, X <: S \in \Gamma$ for some S, and $\mathsf{is\text{-}reader}_{\Gamma, x :_D P, \Delta}\ C\ S$. Now we inspect where $X <: S$ is bound, wherein in each case we can show that $X <: \theta S \in \theta\Gamma, \theta\Delta$. Now we conclude by using the IH and the same rule.

□

Lemma A.51 (Term substitution preserves is-reader). *If (i)* $\mathit{is\text{-}reader}_{\Gamma, x:P, \Delta}\ z$, *and (ii)* $\Gamma \vdash x' : P$, *then* $\mathit{is\text{-}reader}_{\Gamma,\theta\Delta}\ \theta z$, *where* $\theta = [x \mapsto x']$.

Proof. We have $z : T \in \Gamma, x : P, \Delta$ for some T and $\Gamma, x : P, \Delta \vdash T <: \mathsf{Rdr}[S] \wedge C$ for some C, S. We first invoke Lemma A.9 to show that $\Gamma, x : P, \Delta \vdash T \uparrow \mathsf{Rdr}$. Using Lemma A.50 we can show that $\mathsf{is\text{-}reader}_{\Gamma,\theta\Delta}\ \theta T$. Then we perform a case analysis on where z is bound. If $z = x$ then we have $\theta z = x'$ and $\Gamma, \theta\Delta \vdash \theta P \uparrow \mathsf{Rdr}$. By induction on the derivation of $\Gamma \vdash x' : P$ we can show that there exists $S' \wedge C'$ such that $x' : S' \wedge C' \in \Gamma$ and $\Gamma \vdash S' \wedge \{x'\} <: P$. By the well-formedness of the environment, we can show that $\theta P = P$, implying that $\Gamma, \theta\Delta \vdash P \uparrow \mathsf{Rdr}$. Now, by weakening and Lemma A.8, we can show that $\Gamma, \theta\Delta \vdash S' \wedge \{x'\} \uparrow \mathsf{Rdr}$. Finally, we use Lemma A.7 to show that $\Gamma, \theta\Delta \vdash S' \wedge C' \uparrow \mathsf{Rdr}$, and then use Lemma A.9 again to conclude this case. If z is bound in Γ or Δ, in both cases we can show that $z : \theta T \in \Gamma, \theta\Delta$ and therefore conclude directly. □

Lemma A.52 (Term substitution preserves subcapturing). *If (i)* $\Gamma, x :_D P, \Delta \vdash C_1 <: C_2$, *and (ii)* $\Gamma \vdash x' : P$ *then* $\Gamma, \theta\Delta \vdash \theta C_1 <: \theta C_2$ *where* $\theta = [x \mapsto x']$.

Proof. By induction on the subcapture derivation.

Case (SC-TRANS). By applying the IH twice and use the same rule.

Case (SC-ELEM). Then $C_1 = \{y\}$, and $y \in C_2$. If $y \neq x$, we can show that $x \in \theta C_2$, and therefore conclude the case by (SC-ELEM) again. Otherwise, if $y = x$, we can show that $x' \in \theta C_2$. This case is therefore conclude by (SC-ELEM).

Case (SC-SET). Then $\overline{\Gamma, x :_D P, \Delta \vdash \{x\} <: C_2}^{x \in C_1}$. By applying the IH repeatedly, we can show that $\overline{\Gamma, \theta\Delta \vdash \theta\,\{x\} <: \theta C_2}^{x \in C_1}$. Note that $\theta C_1 = \bigcup_{x \in C_1} \theta\,\{x\}$. This case is therefore concluded by (SC-SET).

Case (SC-VAR). Then $C_1 = \{z\}$, and $z : S \wedge C_2 \in \Gamma, x :_D P, \Delta$. Now we inspect where z is bound.

- *When $x = z$.* The goal becomes $\Gamma, \theta\Delta \vdash x' <: \theta C_2$. By inspecting the derivation of $\Gamma \vdash x' : P$, we can show that $x' : P' \in \Gamma$ and $\Gamma \vdash \{x'\} <: C_2$. By weakening we have $\Gamma, \theta\Delta \vdash \{x'\} <: C_2$. Also, by the well-formedness, we can show that $x \notin C_2$ and therefore $\theta C_2 = C_2$. This case is therefore concluded.
- *When z is bound in Γ.* By well-formedness we know that $x \notin \mathrm{fv}(C_2)$. Therefore, $\theta C_2 = C_2$. Also, $z : S \wedge C_2 \in \Gamma, \theta\Delta$, and we conclude this case by (SC-VAR).
- *When z is bound in Δ.* Then $z : \theta C_2 \wedge \theta S \in \Gamma, \theta\Delta$. Therefore this case is concluded by IH and (SC-VAR).

Case (SC-RDR-CAP) and (SC-REF-CAP). By the same rule.

Case (SC-READER). We conclude using the IH, Lemma A.51, and the same rule. □

Lemma A.53 (Term substitution preserves subtyping). *If (i) $\Gamma, x :_D P, \Delta \vdash T <: U$, and (ii) $\Gamma \vdash x' : P$ then $\Gamma, \theta\Delta \vdash \theta T <: \theta U$ where $\theta = [x \mapsto x']$.*

Proof. By induction on the subtype derivation.

Case (REFL) and (TOP). By the same rule.

Case (CAPT). By IH, Lemma A.52 and the same rule.

Case (TRANS), (BOXED), (FUN) and (TFUN). By IH and the same rule.

Case (TVAR). Then $T = X$, $U = S$, and $X <: S \in \Gamma, x :_D P, \Delta$. Our goal is to show that $\Gamma, \theta\Delta \vdash \theta X <: \theta S$. We inspect where X is bound. First, we show that $\theta X = X$ since $X \neq x$. If $X \in \mathrm{dom}(\Gamma)$, we can show that $x \notin \mathrm{fv}(S)$ by the well-formedness of the environment. Therefore $\theta S = S$. Since $X <: S \in \Gamma, \theta\Delta$, we can conclude this case by (TVAR). Otherwise if $X \in \mathrm{dom}(\Delta)$, we have $X <: \theta S \in \Gamma, \theta\Delta$. This case is therefore concluded by (TVAR) too. □

Lemma A.54 (Term substitution preserves separation). *If (i) $\Gamma, x :_D P, \Delta \vdash C_1 \bowtie C_2$, (ii) $\Gamma \vdash x' : P$ and (iii) $\Gamma \vdash D \bowtie x'$, then $\Gamma, \theta\Delta \vdash \theta C_1 \bowtie \theta C_2$ where $\theta = [x \mapsto x']$.*

Proof. By induction on the separation derivation.

Case (NI-SYMM). Then $\Gamma, x :_D P, \Delta \vdash C_2 \bowtie C_1$. We conclude this case by IH and the same rule.

Case (NI-SET). Then $C_1 = \{\tau_i\}_{i=1,\cdots,n}$. By repeated IH we have $\overline{\Gamma, \theta\Delta \vdash \theta\tau_i \bowtie \theta C_2}^{i=1,\cdots,n}$. Note that $\theta C_1 = \bigcup_{i=1,\cdots,n} \theta\tau_i$. we can therefore conclude this case by (NI-SET).

Case (NI-DEGREE). Then $C_1 = \{z_1\}$, $C_2 = \{z_2\}$, $z_1 :_{D_1} T \in \Gamma, x :_D P, \Delta$, and $z_2 \in D_1$. Now we do a case analysis on where z_1 is bound.

- *When $z_1 = x$.* Then we have $\theta C_1 = x'$ and $z_2 \in D$. By the well-formedness of the environment, $x \notin D$, therefore $z_2 \neq x$ and $\theta z_2 = z_2$. The goal becomes $\Gamma, \theta\Delta \vdash x' \bowtie z_2$. Note that we have $\Gamma \vdash D \bowtie x'$, on which we can invoke Lemma A.19 to show that $\Gamma \vdash z_2 \bowtie x'$. Now we conclude by (NI-SYMM) and weakening.
- If $z_1 \in \mathrm{dom}(\Gamma)$. We have $\theta D_1 = D_1$, and $x \notin D_1$ by the well-formedness. Therefore, we have $z_2 \neq x$, which implies that $\theta\,\{z_2\} = \{z_2\}$. Now we can conclude this case by (NI-DEGREE)
- If $z_1 \in \mathrm{dom}(\Delta)$. Then $z_1 :_{\theta D_1} \theta T \in \theta\Delta$. We can show that $\theta z_2 \in \theta D_1$. This case is therefore concluded by IH and (NI-DEGREE).

Case (NI-VAR). Then $C_1 = \{y\}$, $y :_{D_1} T \in \Gamma, x :_D P, \Delta$, and $\Gamma, x :_D P, \Delta \vdash \mathrm{cs}(T) \bowtie C_2$. By the IH we can show that $\Gamma, \theta\Delta \vdash \theta\mathrm{cs}(T) \bowtie \theta C_2$. Now we inspect where y is bound.

- If $x = y$, then $D = D_1$ and $T = P$. The goal becomes $\Gamma, \theta\Delta \vdash x' \bowtie \theta C_2$. By inspecting the derivation of $\Gamma \vdash x' : P$, we can show that $\Gamma \vdash \{x'\} <: \mathrm{cs}(P)$. By the well-formedness we can show that $\theta\mathrm{cs}(P) = \mathrm{cs}(P)$. Now we can conclude this case by Lemma A.22.
- If $y \in \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Delta)$, we can show that in both cases $y :_{\theta D_1} \theta T \in \Gamma, \theta D$. We can conclude immediately by using the IH and the (NI-VAR) rule.

Case (NI-READER). Then $C_1 = \{z_1\}$, $C_2 = \{z_2\}$, and $\Gamma \vdash \{z_i\} <: \{\mathbf{rdr}\}$ for $i = 1, 2$. We conclude by using Lemma A.52. □

Lemma A.55 (Term substitution preserves typing). *If (i)* $\Gamma, x :_D P, \Delta \vdash t : T$, *(ii)* $\Gamma \vdash x' : P$ *and (iii)* $\Gamma \vdash D \bowtie x'$, *then* $\Gamma, \theta\Delta \vdash \theta t : \theta T$ *where* $\theta = [x \mapsto x']$.

Proof. By induction on the typing derivation.

Case (VAR). Then $t = y$ and $y :_{D_y} T \in \Gamma, x :_D P, \Delta$. If $x = y$, we have $T = P$ and the goal is to show that $\Gamma, \theta\Delta \vdash x' : \theta P$. By the well-formedness of $\Gamma, x :_D P$, $x \notin \theta P$. Therefore, $\theta P = P$ and we conclude this case from the premise. Otherwise, if $x \neq y$, we have $\theta t = y$, and we proceed by a case analysis on where y is bound.

- If $y \in \mathrm{dom}(\Gamma)$, then $x \notin T$ by the well-formedness of $\Gamma, x :_D P$, and therefore $\theta T = T$. This case can be concluded by (VAR).
- If $y \in \mathrm{dom}(\Delta)$, then the goal becomes $\Gamma, \theta\Delta \vdash y : \theta T$. We have $y :_{\theta D} \theta T \in \Delta$ and this case is again concluded by (VAR).

Case (SUB). By IH, Lemma A.53, and (SUB).

Case (ABS). Then $t = \lambda(z :_{D_z} U)s$, $T = \mathrm{cv}(s) / z\, \forall(z :_{D_z} U)Q$, and $\Gamma, x :_D P, \Delta, z :_{D_z} U \vdash s : Q$. By IH we show that $\Gamma, x :_D P, \theta\Delta, z :_{\theta D_z} \theta U \vdash \theta s : \theta Q$. This case is therefore concluded by (ABS).

Case (TABS). As above.

Case (APP). Then $t = y_1\, y_2, \Gamma, x :_D P, \Delta \vdash y_1 : C\, \forall(z :_{D_f} U)T'$ where $T = [z \mapsto y_2]\, T'$, $\Gamma, x :_D P, \Delta \vdash y_2 : U$, and $\Gamma, x :_D P, \Delta \vdash D_f \bowtie y_2$. By IH, $\Gamma, \theta\Delta \vdash \theta y_1 : \theta C\, \forall(z :_{\theta D_f} \theta U)\theta T$ and $\Gamma, \theta\Delta \vdash \theta y_2 :_{D'_f} \theta U$. By Lemma A.54, we can show that $\Gamma, \theta\Delta \vdash \theta D_f : \theta y_2$. Now we can invoke (APP-R) to show that $\Gamma, \theta\Delta \vdash \theta(y_1\, y_2) : [z \mapsto \theta y_2]\, (\theta T')$. Since z is fresh we have $\theta([z \mapsto y_2]\, T')$ and conclude this case.

Case (TAPP). As above.

Case (BOX). Then $t = \Box y$, and $\Gamma, x :_D P, \Delta \vdash y : S \wedge C$ and $T = \Box S \wedge C$. By IH, we have $\Gamma, \theta\Delta \vdash \theta y : \theta(S \wedge C)$. Since $\theta C \subseteq \mathrm{dom}(\Gamma, \theta\Delta)$, we can show that $\Gamma, \theta \vdash \theta(\Box y) : \Box\theta(S \wedge C)$ by (BOX) and conclude this case.

Case (UNBOX). As above.

Case (LET). Then $t = \mathrm{let}_m\, z = s$ in u, $\Gamma, x :_{\{\}} P, \Delta \vdash s : U$, $\Gamma, x :_D P, \Delta, z :_{\{\}} U \vdash u : T$, and $\Gamma, x :_D P, \Delta \vdash s \bowtie u$. By IH, we have $\Gamma, \theta\Delta \vdash \theta s : \theta U$, and $\Gamma, \theta\Delta, z :_{\{\}} \theta U \vdash \theta u : \theta T$. By Lemma A.54, we show that $\Gamma, \theta\Delta \vdash \theta s \bowtie \theta u$. Now we can conclude this case by (LET).

Case (DVAR). Then $t = \mathrm{var}_{D_z}\, z := y$ in $s, \Gamma, x :_D P, \Delta \vdash y : S, \Gamma, x :_D P, \Delta, z :_{D_z} \mathrm{Ref}[S] \wedge \{\mathbf{ref}\} \vdash s : T$. We can conclude this case by IH and (DVAR).

Case (READ) and (WRITE). By IH and the same rule. □

Type Substitution.

Lemma A.56 (Type substitution preserves reader checking). *If (i)* $\Gamma, X <: S, \Delta \vdash T \uparrow \mathit{Rdr}$, *and (ii)* $\Gamma \vdash R <: S$, *then is-reader*$_{\Gamma, \theta\Delta}\, \theta T$, *where* $\theta = [X \mapsto R]$.

Proof. By induction on the derivation.

Case (RD-READER). We conclude immediately using the same rule.

Case (RD-TVAR). Then $T = X \wedge C$ for some C, Y, $Y <: S_0 \in \Gamma, X <: S, \Delta$, and $\Gamma, X <: S, \Delta \vdash S_0 \wedge C \uparrow \mathrm{Rdr}$. By the IH we can show that $\Gamma, \theta\Delta \vdash \theta(S_0 \wedge C) \uparrow \mathrm{Rdr}$. Now, we proceed by a case analysis on where Y is bound.

- If $Y = X$. Then $\theta(Y \wedge C) = R \wedge C$ and $S_0 = S$. First, by the wellformedness we can show that $\theta(S_0 \wedge C) = S_0 \wedge C$, and therefore $\Gamma, \theta\Delta \vdash S_0 \wedge C \uparrow \mathrm{Rdr}$. Now we conclude this case by weakening and Lemma A.8.
- If Y is bound in either Γ or Δ then in both cases we have $Y <: \theta S_0 \in \Gamma, \theta\Delta$. We conclude by the IH and the (RD-TVAR) rule.

□

Lemma A.57 (Type substitution preserves is-reader). *If (i) $\textit{is-reader}_{\Gamma, X <: S, \Delta}\ z$, and (ii) $\Gamma \vdash R <: S$, then $\textit{is-reader}_{\Gamma, \theta\Delta}\ \theta z$, where $\theta = [X \mapsto R]$.*

Proof. Then we have $z : T \in \Gamma, X <: S, \Delta$, and $\Gamma, X <: S, \Delta \vdash T <: \mathrm{Rdr}[S_0] \wedge C_0$ for some C_0, S_0. We first invoke Lemma A.8 to show that $\Gamma, X <: S, \Delta \vdash T \uparrow \mathrm{Rdr}$. Then, by Lemma A.56 we can show that $\Gamma, \theta\Delta \vdash \theta T \uparrow \mathrm{Rdr}$. By inspecting where z is bound, we can show that $z : \theta T \in \Gamma, \theta\Delta$, and finally conclude by Lemma A.9 and the (RD-TVAR) rule.

□

Lemma A.58 (Type substitution preserves subcapturing). *If (i) $\Gamma, X <: S, \Delta \vdash C <: D$, and (ii) $\Gamma \vdash R <: S$, then $\Gamma, \theta\Delta \vdash C : D$, where $\theta = [X \mapsto R]$.*

Proof. By induction on the subcapture derivation, wherein most cases do not rely on the type bindings in the context and thus can be concluded immediatley by the IH and the same rule. In the (SC-READER) case we invoke Lemma A.57 to conclude. □

Lemma A.59 (Type substitution preserves subtyping). *If (i) $\Gamma, X <: S, \Delta \vdash T <: U$, and (ii) $\Gamma \vdash R <: S$, then $\Gamma, \theta\Delta \vdash \theta T <: \theta U$, where $\theta = [X \mapsto R]$.*

Proof. By induction on the subtype derivation.
Case (REFL) and (TOP). By the same rule.
Case (CAPT). By Lemma A.58, and (CAPT).
Case (TRANS), (BOXED), (FUN) and (TFUN). By IH and application of the same rule.
Case (TVAR). Then $T = Y$, $Y <: S' \in \Gamma, X <: S, \Delta$, and $U = S'$. Our goal is $\Gamma, \theta\Delta \vdash \theta Y <: \theta S'$. Now we inspect where Y is bound.

- *When $X = Y$.* Then $S' = S$, and the goal becomes $\Gamma, \theta\Delta \vdash R <: \theta S$. By the well-formedness, we can show that $X \notin \mathrm{fv}(S)$ and therefore $\theta S = S$. Now we conclude by weakening the premise.
- *When $Y \in \mathit{dom}(\Gamma)$.* By the well-formedness, $X \notin \mathrm{fv}(S')$. Therefore, $\theta S' = S'$. Also, $\theta Y = Y$, and $Y <: S' \in \Gamma, \theta\Delta$. This case is therefore concluded by (TVAR).
- *When $Y \in \mathit{dom}(\Delta)$.* Then $Y <: \theta S' \in \Gamma, \theta\Delta$. This case is therefore concluded by (TVAR).

□

Lemma A.60 (Type substitution preserves separation). *If (i) $\Gamma, X <: S, \Delta \vdash C_1 \bowtie C_2$, and (ii) $\Gamma \vdash R <: S$, then $\Gamma, \theta\Delta \vdash C_1 \bowtie C_2$, where $\theta = [X \mapsto R]$.*

Proof. By induction on the non-intereference derivation.
Case (NI-SYMM). Then $\Gamma, X <: S, \Delta \vdash C_2 \bowtie C_1$. We conclude this case by the IH and (NI-SYMM).
Case (NI-SET). Then $\overline{\Gamma, X <: S, \Delta \vdash y \bowtie C_2}^{\,y \in C_1}$. By applying the IH repeated, we can show that $\overline{\Gamma, \theta\Delta \vdash y \bowtie C_2}^{\,y \in C_1}$. This case is therefore concluded by (NI-SET).

Case (NI-DEGREE). Then $C_1 = \{x\}$, $C_2 = \{y\}$, $x :_D T \in \Gamma, X <: S, \Delta$, and $y \in D$. Note that $X \notin D$, therefore we have $\theta D = D$. Therefore, no matter where x is bound, we have $x :_D T' \in \Gamma, \theta\Delta$. This case can therefore be concluded by (NI-DEGREE).

Case (NI-VAR). Then $C_1 = \{x\}$, $C_2 = \{y\}$, $x :_D T \in \Gamma, X <: S, \Delta$, and $\Gamma, X <: S, \Delta \vdash \mathrm{cs}(T) \bowtie y$. Depending on where x is bound, either $x :_D T \in \Gamma, \theta\Delta$, or $x :_{\theta D} \theta T \in \theta\Delta$. Since $X \notin \mathrm{cs}(T)$, we can show that $\mathrm{cs}(\theta T) = \theta\mathrm{cs}(T) = \mathrm{cs}(T)$. By IH, we can show that $\Gamma, \theta\Delta \vdash \mathrm{cs}(T) \bowtie y$. We can therefore conclude this case by (NI-VAR).

Case (NI-READER). By applying the IH, Lemma A.58 and the same rule. □

Lemma A.61 (Type substitution preserves typing). *If (i)* $\Gamma, X <: S, \Delta \vdash t : T$, *and (ii)* $\Gamma \vdash R <: S$, *then* $\Gamma, \theta\Delta \vdash \theta t : \theta T$ *where* $\theta = [X \mapsto R]$.

Proof. By induction on the typing derivation.

Case (VAR). Then $t = x$, $x : S' \wedge C \in \Gamma, X <: S, \Delta$, and the goal becomes $\Gamma, \theta\Delta \vdash \theta x : \theta(S' \wedge \{x\})$. Since $x \neq X$, we have $\theta x = x$. Now we inspect where x is bound.

- *When* $x \in dom(\Gamma)$. By the well-formedness we can show that $X \notin \mathrm{fv}(S' \wedge C)$. Therefore, $\theta S' = S'$. We have $x : S' \wedge C \in \Gamma, \theta\Delta$. This case is concluded by (VAR).
- *When* $x \in dom(\Delta)$. We have $x : \theta(S' \wedge C) \in \Gamma, \theta\Delta$. This case is therefore concluded by (VAR).

Case (ABS) and (TABS). In both cases we conclude by IH and the same rule.

Case (APP-R). Then $t = x\, y$, $\Gamma, X <: S, \Delta \vdash x : C \forall(z :_D U) T'$, $\Gamma, X <: S, \Delta \vdash y : U$, $\Gamma, X <: S, \Delta \vdash D \bowtie y$, and $T = [z \mapsto y]\, T'$. We can conclude this case by IH, Lemma A.60, the fact that $\theta\,[z \mapsto y]\, T' = [z \mapsto y]\, \theta T'$ and the (APP-R) rule.

Case (TAPP). Then $t = x[S']$, $\Gamma, X <: S, \Delta \vdash x : C \forall[Z <: S'] T'$ and $T = [Z \mapsto S']\, T'$. The goal becomes $\Gamma, \theta\Delta \vdash \theta(x[S']) : \theta\,[Z \mapsto S']\, T'$. By IH we have $\Gamma, \theta\Delta \vdash \theta x : \theta C \forall[Z <: \theta S'] \theta T'$. By (TAPP) we can show that $\Gamma, \theta\Delta \vdash \theta(x[S']) : [Z \mapsto \theta S']\, \theta T'$. We observe that $[Z \mapsto \theta S']\, \theta T' = \theta\,[Z \mapsto S']\, T'$ and conclude this case.

Case (BOX). Then $t = \Box\, y$ $\Gamma, X <: S, \Delta \vdash y : S' \wedge C$, and $C \subseteq \mathrm{dom}(\Gamma, X <: S, \Delta)$. The goal becomes $\Gamma, \theta\Delta \vdash \theta(\Box\, y) : \theta(\Box\, S' \wedge C)$. Note that since $y \neq X$, we have $\theta y = y$. Proof proceeds by inspecting the location of y in the context.

- *When* $y \in dom(\Gamma)$. By the well-formedness we can show that $X \notin \mathrm{fv}(S' \wedge C)$. Therefore, $\theta(S' \wedge C) = S' \wedge C$. We observe that $y : S' \wedge C \in \Gamma, \theta\Delta$, and show that $\Gamma, \theta\Delta \vdash y : S' \wedge C$ by (VAR). Now this case can be concluded by (BOX).
- *When* $y \in dom(\Delta)$. Then we observe that $y : \theta(S' \wedge C) \in \Gamma, \theta\Delta$, and derive that $\Gamma, \theta\Delta \vdash y : \theta(S' \wedge C)$ by (VAR). This case is concluded by (BOX).

Case (UNBOX). Analogous to the previous case.

Case (SUB). By IH, Lemma A.59 and (SUB).

Case (LET). Then $t = \mathrm{let}_m\, x = s$ in u, $\Gamma, X <: S, \Delta \vdash s : T_0$, $\Gamma, X <: S, \Delta, x :_{\{\}} T_0 \vdash u : T$, and $\Gamma, X <: S, \Delta \vdash s \bowtie u$. By IH we can show that $\Gamma, \theta\Delta \vdash \theta s : \theta T_0$, and $\Gamma, \theta\Delta, x :_{\{\}} \theta T_0 \vdash \theta u : \theta T$. Since $X \notin \mathrm{cv}(s), \mathrm{cv}(u), \mathrm{cs}(T_0)$, we can show that $\theta\mathrm{cv}(s) = \mathrm{cv}(s)$, and $\theta\mathrm{cv}(u) = \mathrm{cv}(u)$. By Lemma A.60, we can show that $\Gamma, \theta\Delta \vdash \theta s \bowtie \theta u$. We can therefore conclude this case by (LET).

Case (DVAR). Then $t = \mathrm{var}\, x = y$ in s, $\Gamma, X <: S, \Delta \vdash y : S$, $\Gamma, X <: S, \Delta, x :_D \mathrm{Ref}[S] \wedge \{\mathbf{ref}\} \vdash s : T$, and $\Gamma, X <: S, \Delta \vdash D$ **wf**. By IH we can show that $\Gamma, \theta\Delta \vdash \theta y : \theta S$, and $\Gamma, \theta\Delta, x :_{\theta D} \theta(\mathrm{Ref}[S] \wedge \{\mathbf{ref}\}) \vdash \theta s : \theta T$. Also, we can show that $\Gamma, \theta\Delta \vdash D$ **wf**. This case is therefore concluded by (DVAR).

Case (READ) and (WRITE). By IH and the same rule. □

A.2.7 Soundness Theorems.

Lemma A.62 (Canonical forms: term abstraction). $\Gamma \vdash v : C \forall(x :_D U) T$ *implies* $v = \lambda(x :_D U') t$ *for some* U' *and* t, *such that* $\Gamma \vdash U <: U'$ *and* $\Gamma, x :_D U' \vdash t : T$.

Proof. By induction on the typing derivation.
Case (ABS). We conclude immediately from the premise.
Case (SUB). Then $\Gamma \vdash v : T_0$, and $\Gamma \vdash T_0 <: C\,\forall(x :_D U)T$. By Lemma A.16 and Lemma A.36, we can show that $T_0 = C_0, \forall(x :_D U_0)T_0$, $\Gamma \vdash U <: U_0$, and $\Gamma, x :_D U_0 \vdash T_0 <: T$. Invoking IH, we can show that $v = \lambda(x :_D U')t$ such that $\Gamma \vdash U_0 <: U'$, and $\Gamma, x :_D U' \vdash t : T_0$. We conclude by the (TRANS) and the (SUB) rule. □

Lemma A.63 (Canonical forms: type abstraction). *$\Gamma \vdash v : C\,\forall[X <: S]T$ implies $v = \lambda[X <: S']t$ for some S' and t, such that $\Gamma \vdash S <: S'$ and $\Gamma, X <: S' \vdash t : T$.*

Proof. Analogous to the proof of Lemma A.62. □

Lemma A.64 (Canonical forms: boxed term). *$\Gamma \vdash v : C\,\square\,T$ implies $v = \square\,x$ for some x, such that $\Gamma \vdash x : T$ and $cs(T) \subseteq dom(\Gamma)$.*

Proof. Analogous to the proof of Lemma A.62. □

Lemma A.65 (Canonical forms: reader). *$\Gamma \vdash v : C\ Rdr[S]$ implies $v = reader\ x$ for some x, such that $\Gamma \vdash x : C\ Ref[S]$.*

Proof. Analogous to the proof of Lemma A.62. □

Lemma A.66 (Store lookup inversion: typing). *If (i) $\vdash \gamma \sim \Gamma$ (ii) $\Gamma, \Delta \vdash x : S \wedge C$, (iii) $\gamma(val\ x) = v$, then $\Gamma, \Delta \vdash v : S \wedge C'$.*

Proof. By induction on the derivation of $\vdash \gamma \sim \Gamma$.
Case (ST-EMPTY). Contradictory.
Case (ST-VAL). Then $\gamma = \gamma_0, \mathrm{val}\ y \mapsto v$, $\vdash \gamma_0 \sim \Gamma_0, \Gamma_0 \vdash v : T$, and $\Gamma = \Gamma_0, y : T$. If $x = y$, then $x : T \in \Gamma, \Delta$. We have $\gamma(\mathrm{val}\ x) = v$. By inspecting the typing derivation $\Gamma, \Delta \vdash x : S \wedge C$, we can show that $\Gamma, \Delta \vdash \{x\} <: C$ and $\Gamma, \Delta \vdash S' <: S$ where $T = S' \wedge C'$ for some C'. By weakening we show that $\Gamma, \Delta \vdash v : S' \wedge C'$. Then, we invoke the (SUB) rule and conclude. Otherwise, we have $x \neq y$. Finally by the IH we conclude this case.
Case (ST-VAR). Then $\gamma = \gamma_0, \mathrm{var}\ y = v$, $\vdash \gamma_0 \sim \Gamma_0, \Gamma_0 \vdash v : S$, and $\Gamma = \Gamma_0, y :_{\mathrm{dom}(\Gamma_0)} \mathrm{Ref}[S] \wedge \{\mathbf{ref}\}$. If $x = y$, it contradicts with $\gamma(\mathrm{val}\ x) = v$. Otherwise, we have $x \neq y$, we conclude this case by the IH.
Case (ST-SET). By IH. □

Lemma A.67 (Store lookup inversion: term abstraction). *If (i) $\vdash \gamma; e \sim \Gamma; \Delta$; (ii) $\Gamma, \Delta \vdash x : C\,\forall(z :_D T)U$ and (iii) $\gamma(val\ x) = \lambda(z :_D T')t$, then $\Gamma, \Delta \vdash T <: T'$ and $\Gamma, \Delta, z : T' \vdash t : U$.*

Proof. We first invoke Lemma A.66 to show that $\Gamma, \Delta \vdash \lambda(z :_D T')t : \forall(z :_D T)U \wedge C'$. By Lemma A.62, we can show that $\Gamma, \Delta \vdash T <: T'$, and $\Gamma, \Delta, z :_D T' \vdash t : U$. We therefore conclude. □

Lemma A.68 (Store lookup inversion: type abstraction). *If (i) $\vdash \gamma; e \sim \Gamma; \Delta$; (ii) $\Gamma, \Delta \vdash x : C\,\forall[X <: S]U$ and (iii) $\gamma(val\ x) = \lambda[X <: S']t$, then $\Gamma, \Delta \vdash S <: S'$ and $\Gamma, \Delta, X <: S' \vdash t : U$.*

Proof. Analogous to the proof of Lemma A.67. □

Lemma A.69 (Store lookup inversion: boxed term). *If (i) $\vdash \gamma; e \sim \Gamma; \Delta$; (ii) $\Gamma, \Delta \vdash x : C'\,\square\,S \wedge C$ and (iii) $\gamma(val\ x) = \square\,y$, then $\Gamma, \Delta \vdash y : S \wedge C$.*

Proof. Analogous to the proof of Lemma A.67. □

Lemma A.70 (Store lookup inversion: reader). *If (i) $\vdash \gamma; e \sim \Gamma; \Delta$; (ii) $\Gamma, \Delta \vdash x : C\ Rdr[S]$ and (iii) $\gamma(val\ x) = reader\ y$, then $\Gamma, \Delta \vdash y : C'\ Ref[S]$.*

Proof. Analogous to the proof of Lemma A.67. □

Lemma A.71 (Value typing with strict capture set). *$\Gamma \vdash v : T$ implies $\exists T'$ such that $\Gamma \vdash v : T'$ and $cs(T') = cv(v)$.*

Proof. By induction on the typing derivation. Only the following cases are applicable.

Case (abs). Then $v = \lambda(x :_D U)t$, $\Gamma, x :_D U \vdash t : T'$, and $T = \mathsf{cv}(t)\,/x\ \forall(z :_D U)T'$. Then $\mathsf{cs}(T) = \mathsf{cv}(v) = \mathsf{cv}(t)\,/x$. This case is therefore concluded.

Case (tabs) and (box). Analogous to (abs).

Case (sub). By IH. □

Lemma A.72 (Value typing inversion: capture set). *$\Gamma \vdash v : T$ implies $\Gamma \vdash cv(v) <: cs(T)$.*

Proof. Analogous to the proof of Lemma A.71. □

Theorem 4.3 (Preservation). *If (i) $\vdash \gamma \sim \Gamma$, (ii) $\Gamma \vdash t : T$, and (iii) $\gamma \mid t \longmapsto \gamma' \mid t'$, then $\exists \Gamma'$ such that (i) $\vdash \gamma' \sim \Gamma'$ and (ii) $\Gamma' \vdash t' : T$.*

Proof. Proceed by case analysis on the reduction derivation.

Case (apply). Then $t = e\,[\,x\,y\,]$, $\gamma(\mathsf{val}\ x) = \lambda(z :_D U)s$, $t' = [z \mapsto y]\,s$, and $\gamma' = \gamma$. By Lemma 4.5, $\exists \Delta, Q$ such that $\Gamma \vdash e \sim \Delta : [Q \mathbin{@} x\,y] \Rightarrow T$ and $\Gamma, \Delta \vdash x\,y : Q'$, where $Q = [z \mapsto y]\,Q'$. Now, by induction on this typing derivation, we prove that (i) $\Gamma, \Delta \vdash x : C\,\forall(z :_D U')Q'$ (ii) $\Gamma, \Delta \vdash y :_{D'} U'$, and (iii) $\Gamma, \Delta \vdash D \bowtie y$. For the (app) case, it is conclude immediately from the preconditions. For the (sub) case, it can be concluded from IH and (sub). Other cases are not applicable. By Lemma A.67, we can show that $\Gamma, \Delta \vdash U' <: U$, and $\Gamma, \Delta, z :_D U \vdash s : Q'$. We invoke Lemma A.55 to show that $\Gamma, \Delta \vdash [z \mapsto y]\,s : [z \mapsto y]\,Q'$. Since $\gamma(\mathsf{val}\ x) = \lambda(z :_D U)s$ and $\vdash \gamma \sim \Gamma$, we have $x : T_x \in \Gamma$ for some T_x where $\mathsf{cs}(T_x) = \mathsf{cv}(s)\,/z$. We can show that $\mathsf{cv}([z \mapsto y]\,s) \subseteq \mathsf{cv}(x\,y) \setminus \{x\} \cup \mathsf{cs}(T_x)$. Now we invoke Lemma A.28 to show that $\Gamma; \Delta \vdash [z \mapsto y]\,s \triangleleft: x\,y$. By evaluation context reification, we show that $\Gamma \vdash e\,[\,[z \mapsto y]\,s\,] : T$ and conclude this case.

Case (tapply). As above, but making use of Corollary A.26.

Case (open). As above, but making use of Lemma A.25.

Case (get). As above.

Case (lift-let). Then $t = e\,[\,\mathsf{let}_m\ x = v\ \mathsf{in}\ s\,]$, $\gamma' = \gamma, \mathsf{val}_m\ x \mapsto v$, $t' = e\,[\,s\,]$, and $\mathsf{fv}(v) \cup D \subseteq \mathsf{dom}(\gamma)$. Now we invoke Lemma 4.5 to show that $\exists \Delta, Q$ such that $\Gamma \vdash e \sim \Delta : [Q \mathbin{@} \mathsf{let}_m\ x = v\ \mathsf{in}\ s] \Rightarrow T$, and $\Gamma, \Delta \vdash \mathsf{let}_m\ x = v\ \mathsf{in}\ s : Q$. By inspecting this typing derivation, we can show that $\Gamma, \Delta \vdash v : U$, $\Gamma, \Delta, x :_{\{\}} U \vdash s : Q$, and $\Gamma, \Delta \vdash v \bowtie s$ if $m = \mathsf{par}$. By Lemma A.71, narrowing, and Lemma A.22, we can show that $\exists U'$ such that $\mathsf{cs}(U') = \mathsf{cv}(v)$, $\Gamma, \Delta \vdash v : U'$, and $\Gamma, \Delta, x :_D U' \vdash s : Q$. Set $\Gamma' = \Gamma, x :_D U'$. We can show that $\vdash \gamma' \sim \Gamma'$ by (st-val). In the next, we show that $\Gamma, \Delta, x :_D U' \vdash \mathsf{cv}(s) <: \mathsf{cv}(\mathsf{let}_D\ x = v\ \mathsf{in}\ s)$. If $x \notin \mathsf{cv}(s)$, we have $\mathsf{cv}(\mathsf{let}_D\ x = v\ \mathsf{in}\ s) = \mathsf{cv}(s) = \mathsf{cv}(s) \setminus \{x\}$. And the sub-goal is proven by the reflexivity of subcapturing. Otherwise, if $x \in \mathsf{cv}(s)$, we have $\mathsf{cv}(\mathsf{let}_D\ x = v\ \mathsf{in}\ s) = \mathsf{cv}(s) \setminus \{x\} \cup \mathsf{cv}(v)$, and $\mathsf{cv}(s) \subseteq \mathsf{cv}(s) \setminus \{x\} \cup \{x\}$. We can show that $\Gamma, \Delta \vdash \{x\} <: \mathsf{cv}(v)$. By the reflexivity of subcapturing and Corollary A.6, we can prove the goal. Since $\mathsf{cv}(v) \subseteq \mathsf{dom}(\gamma)$, we can show that $\Gamma, x :_{\{\}} U'$ is well-formed. Therefore by permutation we have $\Gamma, x :_{\{\}} U', \Delta \vdash \mathsf{cv}(s) <: \mathsf{cv}(\mathsf{let}_D\ x = v\ \mathsf{in}\ s)$. Now we invoke Lemma A.25 to show that $\Gamma, x :_{\{\}} U'; \Delta \vdash s \triangleleft: \mathsf{let}_D\ x = v\ \mathsf{in}\ s$. By Lemma A.44, we can show that $\Gamma, x :_{\{\}} U' \vdash e \sim \Delta : [Q \mathbin{@} \mathsf{let}_D\ x = v\ \mathsf{in}\ s] \Rightarrow T$. Again by permutation, we show that $\Gamma, x :_{\{\}} U', \Delta \vdash s : Q$. We can therefore invoke the reification to show that $\Gamma' \vdash e\,[\,s\,] : T$ and conclude this case.

Case (rename). Then $t = e\,[\,\mathsf{let}_m\ x = y\ \mathsf{in}\ s\,]$, $t' = e\,[\,[x \mapsto y]\,s\,]$, and $\gamma' = \gamma$. We start by invoking Lemma 4.5 to show that $\exists \Delta, Q$ such that $\Gamma \vdash e \sim \Delta : [Q \mathbin{@} \mathsf{let}_m\ x = y\ \mathsf{in}\ s] \Rightarrow T$ and $\Gamma, \Delta \vdash \mathsf{let}_m\ x = y\ \mathsf{in}\ s : Q$. By inspecting this typing derivation, we can show that $\Gamma, \Delta \vdash y : U$ for some U, $\Gamma, \Delta, x :_m U \vdash s : Q$, and $\Gamma, \Delta \vdash y \bowtie s$. Now we invoke Lemma A.55 to show that $\Gamma, \Delta \vdash [x \mapsto y]\,s : [x \mapsto y]\,Q$.

Since $x \notin \mathsf{fv}(Q)$, we have $[x \mapsto y]\, Q = Q$. Note that we have $\mathsf{cv}\,([x \mapsto y]\, s) = \mathsf{cv}\,(\mathsf{let}_D\ x = y\ \mathsf{in}\ s)$. By Corollary A.26, we can show that $\Gamma; \Delta \vdash [x \mapsto y]\, s \mathrel{\triangleleft:} \mathsf{let}_D\ x = y\ \mathsf{in}\ s$. Now we invoke the reification of evaluation context to show that $\Gamma \vdash e\,[\,[x \mapsto y]\, s\,] : T$ and conclude this case.

Case (LIFT-VAR). Then $t = e\,[\,\mathsf{var}\ x = y\ \mathsf{in}\ s\,]$, $\gamma(\mathsf{val}\ y) = v$, $\gamma' = \gamma, \mathsf{var}\ x = v$, and $t' = e\,[\,s\,]$. By Lemma A.49, we show that $\Gamma, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S]\,\hat{}\,\{\mathbf{ref}\} \vdash e\,[\,s\,] : T$. We can show that $\Gamma \vdash \gamma' \sim \Gamma, x :_{\mathsf{dom}(\Gamma)} \mathsf{Ref}[S]\,\hat{}\,\{\mathbf{ref}\}$ by the (ST-VAR) rule. We can therefore conclude this case.

Case (LIFT-SET). Then $t = e\,[\,x := y\,]$, $\gamma(\mathsf{val}\ y) = v$, $t' = e\,[\,v\,]$, and $\gamma' = \gamma, \mathsf{set}\ x = v$. Now we invoke Lemma 4.5 to show that $\exists \Delta, Q$ such that $\Gamma \vdash e \sim \Delta : [Q \mathbin{@} x := y] \Rightarrow T$, and $\Gamma, \Delta \vdash x := y : Q$. By induction on this typing derivation, we can show that $\exists S$ such that $\Gamma, \Delta \vdash x : C\,\mathsf{Ref}[S]$, $\Gamma, \Delta \vdash y : S$, and $\Gamma, \Delta \vdash S <: Q$. By Lemma A.66, we can show that $\Gamma, \Delta \vdash v : S$. By Lemma A.72, we have $\Gamma, \Delta \vdash \mathsf{cv}\,(v) <: \varnothing$. Therefore we have $\Gamma, \Delta \vdash \mathsf{cv}\,(v) <: \mathsf{cv}\,(x := y)$. Now we invoke Lemma A.25 to show that $\Gamma; \Delta \vdash v \mathrel{\triangleleft:} x := y$. Now we reify the evaluation context to show that $\Gamma \vdash e\,[\,v\,] : T$. The remaining to be shown is that $\vdash \gamma' \sim \Gamma$. By inverting the typing judgment $\Gamma, \Delta \vdash x : \mathsf{Ref}[S]\,\hat{}\,C$ and considering that $x \in \mathsf{dom}(\Gamma)$, we can show that $x :_D \mathsf{Ref}[S]\,\hat{}\,C' \in \Gamma$. By induction on $\vdash \gamma \sim \Gamma$, we can show that $\Gamma \vdash v : S$ by the fact that $\gamma(\mathsf{val}\ y) = v$ and weakening. Now $\vdash \gamma' \sim \Gamma$ can be derived by (ST-SET). This case is therefore concluded. □

Lemma A.73 (Evaluation context trampolining). *Consider a reduction derivation $\gamma \mid t \longmapsto \gamma' \mid t'$. We have $\gamma \mid e\,[\,t\,] \longmapsto \gamma' \mid e\,[\,t'\,]$ for any e.*

Proof. By straightforward case analysis on the reduction derivation. No rule makes use of the evaluation context. All cases are analogous, thus we present only the (APPLY) case.

Case (APPLY). Then $t = e'\,[\,x\,y\,]$, $\gamma(\mathsf{val}\ x) = \lambda(z :_D T)s$, $\gamma' = \gamma$, and $t' = e'\,[\,[z \mapsto y]\, s\,]$. None of these preconditions depends on the evaluation context e. We therefore conclude this case by the same rule. □

Theorem A.74 (Progress). *If (i) $\vdash \gamma \sim \Gamma$, (ii) $\Gamma \vdash t : T$, then either t is an answer, or $\exists \gamma', t'$ such that $\gamma \mid t \longmapsto \gamma' \mid t'$.*

Proof. By induction on the typing derivation.

Case (VAR). Then $t = x$. We can conclude this case because x is an answer.

Case (SUB). We directly apply the IH and conclude.

Case (ABS), (TABS), (BOX) and (READER). We conclude these cases because t is a value.

Case (APP). Then $t = x\,y$, $\Gamma \vdash x : C\,\forall(z :_D U)T'$ where $T = [z \mapsto y]\, T'$, and $\Gamma \vdash y : U$. By Lemma A.40, we can show that there exists v such that $\gamma(\mathsf{val}\ x) = v$. By Lemma A.66, we have $\Gamma \vdash v : C\,\forall(z :_D U)T'$. By Lemma A.62, we have $v = \lambda(z :_D U')s$ for some U' and s. Now we conclude by (APPLY).

Case (TAPP). As above.

Case (UNBOX). Then $t = C \multimap x$, $\Gamma \vdash x : \Box\,(S\,\hat{}\,C)$, and $C \subseteq \mathsf{dom}(\Gamma)$. By Lemma A.39, we can show that $\exists v$ such that $\gamma(\mathsf{val}\ x) = v$. By Lemma A.66, we can show that $\Gamma \vdash v : \Box\,(S\,\hat{}\,C)$. By Lemma A.64, we have $z = \Box\, y$ for some y. Now we conclude by (OPEN).

Case (LET). Then $t = \mathsf{let}_m\ x = s\ \mathsf{in}\ u$, $\Gamma \vdash s : T_0$ for some T_0, and $\Gamma, x :_{\{\}} T_0 \vdash s : T$. Proceed by a case analysis on the kind of s.

- If s is a value, we can show that $\mathsf{fv}(s) \subseteq \mathsf{dom}(\Gamma) = \mathsf{dom}(\gamma)$. This allows for the application of the (LIFT-LET) rule.
- If s is a variable, we can conclude by (RENAME).
- Otherwise s is a term. By IH we can show that $\gamma \mid s \longmapsto \gamma' \mid s'$ for some γ' and s'. By Lemma A.73, we can derive that $\gamma \mid \mathsf{let}_D\ x = s\ \mathsf{in}\ u \longmapsto \gamma' \mid \mathsf{let}_D\ x = s'\ \mathsf{in}\ u$ and conclude this case.

Case (DVAR). Then $t = \mathsf{var}\ x = y$ in s, $\Gamma \vdash y \colon S$, and $\Gamma, x :_D \mathsf{Ref}[S] \wedge \{\mathbf{ref}\} \vdash s \colon T$. Using Lemma A.39, we can show that $\gamma(\mathsf{val}\ y) = v$ for some v. This case is therefore concluded by (LIFT-VAR).

Case (READ). Then $t = \mathsf{read}\, x$, and $\Gamma \vdash x \colon \mathsf{Rdr}[S] \wedge C$. By Lemma A.42, we can show that $\exists y.\gamma(\mathsf{val}\ x) = \mathsf{reader}\ y$. Now we invoke Lemma A.65 to show that $\Gamma \vdash y \colon \mathsf{Ref}[S] \wedge C$. By using Lemma A.38 we can show that $\exists v$ such that $\gamma(\mathsf{var}\ y) = v$. Now we conclude this case by (GET).

Case (WRITE). Then $t = x := y$, $\Gamma \vdash x \colon \mathsf{Ref}[S] \wedge C$, and $\Gamma \vdash y \colon S$. By Lemma A.39, we can show that $\exists v$ such that $\gamma(\mathsf{val}\ y) = v$. Now we conclude this case by (LIFT-SET). □

A.3 Confluence

A.3.1 Properties of Evaluation Context Focuses. A focus is one way to split a term t into an evaluation context e and the focused term s such that $t = e\,[\,s\,]$. When evaluating a let-binding, either the bindee or the continuation get reduced, resulting in multiple possible focuses for the same term. Each focus is determined by the part we choose to reduce for each let expression.

Definition A.3 (Subfocus). *We say $t = e_1\,[\,t_1\,]$ is a* subfocus *of $t = e_2\,[\,t_2\,]$, written $e_1\,[\,t_1\,] \preceq e_2\,[\,t_2\,]$, iff $\exists e$ such that $t_1 = e\,[\,t_2\,]$ and $e_2 = e_1\,[\,e\,]$.*

Definition A.4 (Branched focus). *We say $t = e_1\,[\,t_1\,]$ and $t = e_2\,[\,t_2\,]$ are a pair of branched focuses, written $e_1\,[\,t_1\,] \leftrightarrow e_2\,[\,t_2\,]$, iff $\exists e, D, e_1', e_2'$ such that $t = e\,\big[\,let_D\ x = e_1'\,[\,t_1\,]\ in\ e_2'\,[\,t_2\,]\,\big]$, $e_1 = e\,\big[\,let_D\ x = e_1'\,[\,\,]\ in\ e_2'\,[\,t_2\,]\,\big]$, and $e_2 = e\,\big[\,let_D\ x = e_1'\,[\,t_1\,]\ in\ e_2'\,[\,\,]\,\big]$.*

Lemma A.75 (Extending subfocus). *Given two focuses of the same term $t = e_1\,[\,s_1\,] = e_2\,[\,s_2\,]$, we can show that for any e', $e_1\,[\,s_1\,] \preceq e_2\,[\,s_2\,]$ implies $e'\,[\,e_1\,[\,s_1\,]\,] \preceq e'\,[\,e\,[\,2\,]\,]$.*

Proof. Then for some e, we have $s_1 = e\,[\,s_2\,]$, and $e_2 = e_1\,[\,e\,]$ First of all, we can show that $e\,[\,t\,] = e\,[\,e_1\,[\,s_1\,]\,] = e\,[\,e_2\,[\,s_2\,]\,]$, which means that the extended focuses are still for the same term. $s_1 = e\,[\,s_2\,]$ is unaffected by the extension. And $e'\,[\,e_2\,] = e'\,[\,e_1\,[\,e\,]\,]$ is trivial. □

Lemma A.76 (Extending branched focus). *Given two focuses of the same term $t = e_1\,[\,s_1\,] = e_2\,[\,s_2\,]$, for any e' we can show that $e_1\,[\,s_1\,] \leftrightarrow e_2\,[\,s_2\,]$ implies $e'\,[\,e_1\,[\,s_1\,]\,] \leftrightarrow e'\,[\,e_2\,[\,s_2\,]\,]$.*

Proof. Analogous to proof of Lemma A.75. □

Lemma A.77 (Different focuses of the same term). *Given two focuses $e_1\,[\,s_1\,]$ and $e_2\,[\,s_2\,]$ of the same term t, we can show that one of the followings holds:*

(i) $e_1\,[\,s_1\,] \preceq e_2\,[\,s_2\,]$;
(ii) $e_2\,[\,s_2\,] \preceq e_1\,[\,s_1\,]$;
(iii) $e_1\,[\,s_1\,] \leftrightarrow e_2\,[\,s_2\,]$;
(iv) $e_2\,[\,s_2\,] \leftrightarrow e_1\,[\,s_1\,]$.

Proof. We begin by induction on the first evaluation context e_1.
Case $e_1 = [\,]$. Then we can demonstrate that $e_1\,[\,s_1\,] \preceq e_2\,[\,c_2\,]$ by setting $e = e_2$.
Case $e_1 = let_D\ x = e_1'\ in\ u$. We proceed the proof by induction on e_2.

- *Case* $e_2 = [\,]$. Then we conclude by showing that $e_2\,[\,s_2\,] \preceq e_1\,[\,s_1\,]$.
- *Case* $e_2 = let_m\ x = e_2'\ in\ u$. We can invoke IH and use Lemma A.75 and Lemma A.76 to conclude this case.
- *Case* $e_2 = let_m\ x = u'\ in\ e_2'$. Then we can show that $u' = e_1'\,[\,s_1\,]$ and $u = e_2'\,[\,s_2\,]$. We can therefore conclude this case immediately by showing that $e_1\,[\,s_1\,] \leftrightarrow e_2\,[\,s_2\,]$.

Case $e_1 = let_m\ x = u\ in\ e_1'$. Analogous to the previous case. They are symmetric. □

A.3.2 Confluence Theorems. Now we demonstrate that the reduction of the calculus is *confluent*. This means that though we can arbitrarily interleave the reduction of both sides of the let bindings, the result of the program will always be the same, which implies the absence of data races and thus achieves the safe concurrency guarantee.

First of all, in different reduction paths, the store bindings can be lifted to the store in different orders. For example, we may introduce variables in different orders, or have the set-bindings arranged in different ways. Regardless of these permutations, the respective stores should possess the same interpretations. Thus, to carry out the proof we have consider the equivalence between the stores up to the permutations. To this end, we define the *equivalence* between two evaluation stores up to permutation.

Definition A.5 (Equivalent stores). *We say two stores γ_1 and γ_2 are equivalent, written $\gamma_1 \cong \gamma_2$, iff*

(1) γ_2 is permuted from γ_1;
(2) $\forall x \in bvar(\gamma_1), \gamma_1(var\, x) = \gamma_2(var\, x)$.

Definition A.6 (Equivalent configurations). *We say two configurations $\gamma_1 \mid t_1$ and $\gamma_2 \mid t_2$ are equivalent iff $\gamma_1 \cong \gamma_2$ and $t_1 = t_2$.*

Fact A.78 (Store equivalence is an equivalence relation). *The equivalence between stores $\gamma_1 \cong \gamma_2$ is an equivalence relation: it is reflexive, symmetric and transitive.*

Lemma A.79 (Value lookup in equivalent stores). *If $\gamma_1 \cong \gamma_2$ then $\gamma_1(val\, x) = \gamma_2(val\, x)$.*

Proof. By the definition of store equivalence and the lookup function. The result of the lookup function does not rely on the order of bindings. □

Lemma A.80 (Variable lookup in equivalent stores). *If $\gamma_1 \cong \gamma_2$, then $\gamma_1(var\, x) = \gamma_2(var\, x)$.*

Proof. This follows directly from the definition of store equivalence. □

Lemma A.81 (Typing implies separation). *Given non-value terms s_1 and s_2, $\Gamma \vdash let_{par}\ x = e_1\,[\,s_1\,]$ in $e_2\,[\,s_2\,] : T$ implies $\Gamma \vdash s_1 \bowtie s_2$.*

Proof. By induction on the typing derivation.

Case (LET). Then we have $\Gamma \vdash e_1\,[\,s_1\,] \bowtie e_2\,[\,s_2\,]$. Since neither s_1 nor s_2 is a value, by straightforward induction on e_1 and e_2 we can show that $\mathsf{cv}(s_1) \cap \mathsf{dom}(\Gamma) \subseteq \mathsf{cv}(e_1\,[\,s_1\,]) \cap \mathsf{dom}(\Gamma)$ and $\mathsf{cv}(s_2) \cap \mathsf{dom}(\Gamma) \subseteq \mathsf{cv}(e_2\,[\,s_2\,]) \cap \mathsf{dom}(\Gamma)$. Now we can conclude by applying Corollary A.23.

Case (SUB). By the IH. □

Lemma A.82 (Reference root is separated only from empty captures). *If $\Gamma \vdash C_1 \bowtie C_2$, then (i) $\mathbf{ref} \in C_1$ implies $\Gamma \vdash C_2 <: \{\}$, and (ii) $\mathbf{ref} \in C_2$ implies $\Gamma \vdash C_1 <: \{\}$.*

Proof. By induction on the derivation.

Case (NI-SYMM). By swapping the two conclusions in the IH.

Case (NI-SET). Then $\overline{\Gamma \vdash x \bowtie C_2}^{\,x \in C_1}$. By applying the IH repeatedly, we can show that for any $x \in C_1$, we have (1) $\mathbf{ref} \in \{x\}$ implies $\Gamma \vdash C_2 <: \{\}$, and (2) $\mathbf{ref} \in C_2$ implies $\Gamma \vdash \{x\} <: \{\}$. First, if $\mathbf{ref} \in C_1$, then exists $x \in C_1$ such that $\mathbf{ref} \in \{x\}$, which implies that $\Gamma \vdash C_2 <: \{\}$ by the IH. Besides, if $x \in C_2$, we can show that $\Gamma \vdash \{x\} <: \{\}$ for any $x \in C_1$. By invoking Lemma A.4 repeatedly we can show that $\Gamma \vdash C_1 <: \{\}$. We can therefore conclude this case.

Case (NI-DEGREE). Then $C_1 = \{x\}$ and $C_2 = \{y\}$. We can show that $x \neq \mathbf{ref}$ and $y \neq \mathbf{ref}$ by definition, and thus conclude this case.

Case (NI-VAR). Then $C_1 = \{x\}$, $C_2 = \{y\}$, $x : T \in \Gamma$, and $\Gamma \vdash \mathrm{cs}(T) \bowtie y$. Firstly, we have $x \neq \mathbf{ref}$ and therefore $\mathbf{ref} \notin \{x\}$. Secondly, if $* \in \{y\}$ then by the IH we can show that $\Gamma \vdash \mathrm{cs}(T) <: \{\}$. Now we conclude this case using the (SC-VAR) rule.

Case (NI-READER). Then $C_1 = \{x\}$, $C_2 = \{y\}$, $\Gamma \vdash \{x\} <: \{\mathbf{rdr}\}$, and $\Gamma \vdash \{y\} <: \{\mathbf{rdr}\}$. By straightforward induction on $\Gamma \vdash \{x\} <: \{\mathbf{rdr}\}$ we can show that $\mathbf{ref} \notin \{x\}$ and similarly for $\mathbf{ref} \notin \{y\}$. We can therefore conclude this case. □

Lemma A.83 (A variable reference interferes with itself). *Given any $x :_D S \wedge \{\mathbf{ref}\} \in \Gamma$, $\Gamma \vdash \{x\} \bowtie \{x\}$ is impossible.*

Proof. By induction on the derivation of $\Gamma \vdash \{x\} \bowtie \{x\}$, wherein we derive contradiction for each case.

Case (NI-SYMM) and (NI-SET). By the IH.

Case (NI-DEGREE). Then $x \in D$. This is contradictory with the well-formedness.

Case (NI-VAR). Then $\Gamma \vdash \{\mathbf{ref}\} \bowtie x$. By Lemma A.82 we can show that $\Gamma \vdash \{x\} <: \{\}$. By straightforward induction on it we can derive the contradiction.

Case (NI-READER). Then $\Gamma \vdash \{x\} <: \{\mathbf{rdr}\}$. By straightforward induction on this derivation, we can derive the contradiction too. □

Lemma A.84 (A variable reference interferes with its reader). *In an inert environment Γ, given any $x :_{D_x} Rdr[S] \wedge \{y\} \in \Gamma$, and $y :_{D_y} S' \wedge \{\mathbf{ref}\} \in \Gamma$, $\Gamma \vdash \{x\} \bowtie \{y\}$ is impossible.*

Proof. By induction on the derivation of $\Gamma \vdash \{x\} \bowtie \{y\}$.

Case (NI-SYMM) and (NI-SET). By the IH.

Case (NI-DEGREE). Then $x \in D_y$ or $y \in D_x$, where D_x and D_y denotes the separation degree of x and y in the environment respectively. Since the type of x mentions y, by the wellformedness of the environment, x must be defined after y. Therefore we have $x \notin D_y$. If $y \in D_x$, then by the inertness we can show that $\Gamma \vdash \{y\} \bowtie \{y\}$. Now we derive the contradiction via Lemma A.83.

Case (NI-VAR). Then either $\Gamma \vdash \{\mathbf{ref}\} \bowtie \{x\}$, or $\Gamma \vdash \{y\} \bowtie \{y\}$. In the first case we can show that $\Gamma \vdash \{x\} <: \{\}$ by Lemma A.82, from which we can derive the contradiction. In the second case, we invoke Lemma A.83 to derive the contradiction.

Case (NI-READER). Then we have $\Gamma \vdash \{y\} <: \{\mathbf{rdr}\}$. By induction on the derivation of it, we can show contradiction in each case. □

Corollary A.85. *If (i) $\Gamma \vdash \{x\} \bowtie \{y\}$, (ii) $x : S_1 \wedge \{\mathbf{ref}\} \in \Gamma$ and (iii) $y : S_2 \wedge \{\mathbf{ref}\} \in \Gamma$, then we have $x \neq y$.*

Corollary A.86. *If (i) $\Gamma \vdash \{x\} \bowtie \{y\}$, (ii) $x : Rdr[S_1] \wedge \{z\} \in \Gamma$ and (iii) $y : S_2 \wedge \{\mathbf{ref}\} \in \Gamma$, then $z \neq y$.*

Theorem 4.8 (Diamond property of reduction). *Given two equivalent configurations $\gamma_1 \mid t \equiv \gamma_2 \mid t$, if (1) $\gamma_1 \vdash t$ and $\gamma_2 \vdash t$; (2) $\gamma_1 \mid t \longmapsto \gamma_1' \mid t_1$; and (3) $\gamma_2 \mid t \longmapsto \gamma_2' \mid t_2$, then either $t_1 = t_2$, or there exists $\gamma_1'', t', \gamma_2''$ such that (1) $\gamma_1' \mid t_1 \longmapsto \gamma_1'' \mid t'$, (2) $\gamma_2' \mid t_2 \longmapsto \gamma_2'' \mid t'$, and (3) $\gamma_1'' \mid t' \equiv \gamma_2'' \mid t'$.*

Proof. Begin with a case analysis on the derivation of $\gamma_1 \mid t \longmapsto \gamma_1' \mid t_1$.

Case (APPLY), (TAPPLY) and (OPEN). Proceed by a case analysis on the derivation of $\gamma_2 \mid t \longmapsto \gamma_2' \mid t_2$.

- *Case (APPLY), (TAPPLY), (OPEN) and (GET).* We only present the proof when both derivations are derived by the (APPLY) case, and all other possibilities follow analogously. Firstly, from the preconditions we know that $t = e_1\ [\ x_1\ y_1\ ]$ for some $e_1, x_1, y_1, \gamma_1(\mathrm{val}\ x_1) = \lambda(z : T_1)s_1$, and $t_1' = e_1\ [\ [z \mapsto y_1]\ s_1\ ]$. Also, we have $t = e_2\ [\ x_2\ y_2\ ]$ for some $e_2, x_2, y_2, \gamma_2(\mathrm{val}\ x_2) = \lambda(z : T_2)s_2$, and $t_2' = e_2\ [\ [z \mapsto y_2]\ s_2\ ]$. By invoking Lemma A.77, we can show that either $e_1 = e_2$ and $x_1\ y_1 = x_2\ y_2$, or $e_1\ [\ x_1\ y_1\ ] \leftrightarrow e_2\ [\ x_2\ y_2\ ]$, or $e_2\ [\ x_2\ y_2\ ] \leftrightarrow e_1\ [\ x_1\ y_1\ ]$.

  - When $e_1 = e_2$, $x_1 = x_2$ and $y_1 = y_2$.
  - When $e_1\ [\ x_1\ y_1\ ] \leftrightarrow e_2\ [\ x_2\ y_2\ ]$. Then $t = e\ [\ \mathsf{let}_D\ x = e_1\ [\ x_1\ y_1\ ]\ \mathsf{in}\ e_2\ [\ x_2\ y_2\ ]\ ]$. Therefore, we have $t_1 = e\ [\ \mathsf{let}_D\ x = e_1\ [\ [z \mapsto y_1]\ s_1\ ]\ \mathsf{in}\ e_2\ [\ x_2\ y_2\ ]\ ]$. We let $t' = e[\ \mathsf{let}_D\ x = e_1\ [\ [z \mapsto y_1]\ s_1\ ]\ \mathsf{in}\ e_2\ [\ e_2\ [\ [z \mapsto y_2]\ s_2\ ]\ ]\ ]$, and can derive that $\gamma_1 \mid t_1 \longmapsto \gamma_1 \mid t'$. Similarly, we can show that $\gamma_2 \mid t_2 \longmapsto \gamma_2 \mid t'$, and thus conclude by the fact that $\gamma_1 \cong \gamma_2$.
  - When $e_2\ [\ x_2\ y_2\ ] \leftrightarrow e_1\ [\ x_1\ y_1\ ]$. Analogous to the previous case.
- *Case (LIFT-LET).* We only present the proof of the (APPLY) case, and other cases follows analogously. Then $t = e_1\ [\ x_1\ y_1\ ]$ for some e_1, x_1, y_1, $\gamma_1(\mathsf{val}\ x_1) = \lambda(z : T_1)s_1$, and $t_1 = e_1\ [\ [z \mapsto y_1]\ s_1\ ]$. Also, $t = e_2\ [\ \mathsf{let}_D\ x = v\ \mathsf{in}\ u\ ]$, $\gamma_2' = \gamma_2, \mathsf{val}\ x \mapsto v$, and $t_2 = e_2\ [\ u\ ]$. Now we proceed the proof by invoking Lemma A.77 to analyze the relationship of two focuses $t = e_1\ [\ x_1\ y_1\ ] = e_2\ [\ \mathsf{let}_D\ x = v\ \mathsf{in}\ u\ ]$.
  - *When* $e_1\ [\ x_1\ y_1\ ] \preceq e_2\ [\ \mathit{let}_m\ x = v\ \mathit{in}\ u\ ]$. This implies that for some e we have $x_1\ y_1 = e\ [\ \mathsf{let}_m\ x = v\ \mathsf{in}\ u\ ]$, which is impossible.
  - *When* $e_2\ [\ \mathit{let}_m\ x = v\ \mathit{in}\ u\ ] \preceq e_1\ [\ x_1\ y_1\ ]$. Then for some e we have $\boxed{\mathsf{let}_m\ x = v\ \mathsf{in}\ u} = e\ [\ x_1\ y_1\ ]$ and $e_1 = e_2\ [\ e\ ]$. By analyzing the equality, we can show that $t = e_2[\mathsf{let}_m\ x = v\ \mathsf{in}\ e'\ [\ x_1\ y_1\ ]]$ and $e_1 = e_2\ [\ \mathsf{let}_m\ x = v\ \mathsf{in}\ e'\ ]$. Set $t' = e_2\ [\ e'\ [\ [z \mapsto y_1]\ s_1\ ]\ ]$. First, we can derive that $\gamma_1 \mid e_1\ [\ [z \mapsto y_1]\ s_1\ ] \longmapsto \gamma_1, \mathsf{val}\ x \mapsto v \mid e_2\ [\ e'\ [\ [z \mapsto y_1]\ s_1\ ]\ ]$ by showing that $\mathsf{fv}(v) \subseteq \mathsf{dom}(\gamma_1)$ and then apply the (LIFT-LET) rule. Then, we can derive that $\gamma_2, \mathsf{val}\ x \mapsto v \mid e_2\ [\ e'\ [\ x_1\ y_1\ ]\ ] \longmapsto \gamma_2, \mathsf{val}_D\ x \mapsto v \mid e_2\ [\ e'\ [\ [z \mapsto y_1]\ s_1\ ]\ ]$ by applying Lemma A.79 and the (APPLY) rule. Now we can show that $(\gamma_1, \mathsf{val}\ x \mapsto v) \cong (\gamma_2, \mathsf{val}\ x \mapsto v)$ and conclude this case.
  - *When* $e_1\ [\ x_1\ y_1\ ] \leftrightarrow e_2\ [\ \mathit{let}_m\ x = v\ \mathit{in}\ u\ ]$. Then $t = e[\mathsf{let}_n\ y = e_1'\ [\ x_1\ y_1\ ]\ \mathsf{in}\ e_2'[\mathsf{let}_m\ x = v\ \mathsf{in}\ u]]$ for some e, n, y, e_1' and e_2'. Let $\gamma_1'' = \gamma_1, \mathsf{val}_m\ x \mapsto v$, $\gamma_2'' = \gamma_2, \mathsf{val}_m\ x \mapsto v$, and $t' = e\ \big[\ \mathsf{let}_E\ y = e_1'\ [\ [z \mapsto y_1]\ s_1\ ]\ \mathsf{in}\ e_2'\ [\ u\ ]\ \big]$. For both t_1, t_2, we can invoke the other rule to reduce to t'. We can therefore conclude this case.
  - *When* $e_2\ [\ \mathit{let}_m\ x = v\ \mathit{in}\ u\ ] \leftrightarrow e_1\ [\ x_1\ y_1\ ]$. Analogous to the above case. These two cases are symmetric.
- *Case (RENAME), (LIFT-VAR) and (LIFT-SET).* Analogous to the previous case.

Case (GET). Then $t = e_1\ [\ \mathsf{read}\ x\ ]$, $t_1 = e_1\ [\ v\ ]$, $\gamma(\mathsf{val}\ x) = \mathsf{reader}\ x'$ and $\gamma(\mathsf{var}\ x') = v$. We proceed the proof by case analysis on the second derivation, wherein the (APPLY), (TAPPLY) and the (OPEN) cases are symmetric to the above proven cases. The proof of (LIFT-LET), (LIFT-VAR) and (RENAME) is analogous to the previous case. Notably, in the (LIFT-VAR) we are sure that the lifted variable is not x since it is a freshly created local variable.

- *Case (GET).* To prove this case, we begin by invoking Lemma A.77 to analyze the relation between two focuses, wherein in each possibility we can invoke Lemma A.80 to demonstrate the equality between the result of variable lookup in two equivalent stores, and therefore invoke (GET) again to reduce both sides to the same term.
- *Case (LIFT-SET).* Then $t = e_2\ [\ y_1 := y_2\ ]$, $\gamma_2(\mathsf{val}\ y_2) = w$, $\gamma_2' = \gamma_2, \mathsf{set}\ y_1 = w$, and $t_2 = e_2\ [\ w\ ]$. Now we have to show that $y_1 \neq x'$ so that the read and the write can be swapped. We first invoke Lemma A.77 to analyze the relation between two focuses. The two cases where $e_1\ [\ \mathsf{read}\ x\ ] \preceq e_2\ [\ y_1 := y_2\ ]$ or $e_2\ [\ y_1 := y_2\ ] \preceq e_1\ [\ \mathsf{read}\ x\ ]$ are impossible. Now we show the proof when $e_1\ [\ \mathsf{read}\ x\ ] \leftrightarrow e_2\ [\ y_1 := y_2\ ]$, whereas the other case is analogous. We have $t = e\ \big[\ \mathsf{let}_m\ x = e_1'\ [\ \mathsf{read}\ x\ ]\ \mathsf{in}\ e_2'\ [\ y_1 := y_2\ ]\ \big]$. Let $t' = e\ \big[\ \mathsf{let}_m\ x = e_1'\ [\ v\ ]\ \mathsf{in}\ e_2'\ [\ w\ ]\ \big]$. By invoking Lemma A.79 we can show that $\gamma_1(\mathsf{val}\ y_2) = w$, and therefore derive that $\gamma_1 \mid t_1 \longmapsto \gamma_1, \mathsf{set}\ y_1 = w \mid t'$ by the (LIFT-SET) rule. Since $\gamma_2 \vdash t$ we know that $\Gamma_2 \vdash t : T_2$ for some Γ_2 and T_2. By Lemma 4.5 we can show that $\Gamma_2, \Delta_2 \vdash \mathsf{let}_m\ x = e_1'\ [\ \mathsf{read}\ x\ ]\ \mathsf{in}\ e_2'\ [\ y_1 := y_2\ ] : U$ for

some Δ_2 and U. Now we invoke Lemma A.81 and Lemma A.19 to show that $\Gamma_2, \Delta_2 \vdash x \bowtie y_1$. By induction on $\vdash \gamma \sim \Gamma_2$, we can show that $x : \{x'\}\ \mathsf{Rdr}[S_0] \in \Gamma_2$, and $y_1 : \{*\}\ \mathsf{Ref}[S_1] \in \Gamma_2$ for some S_0, S_1. Now we invoke Corollary A.86 to show that $x' \neq y_1$. Therefore, we can show that $\gamma_2'(\mathsf{var}\ x') = \gamma(\mathsf{var}\ x') = v$ where $\gamma_2' = \gamma_2, \mathsf{set}\ y_1 = w$. We can thus derive that $\gamma_2' \mid t_2 \longmapsto \gamma_2' \mid t'$. Finally, we can show that $\gamma_1, \mathsf{set}\ y_1 = w$ and $\gamma_2, \mathsf{set}\ y_1 = w$ are still equivalent and conclude this case.

Case (RENAME). Then $t = e_1 \left[\mathsf{let}_{D_1}\ x_1 = y_1\ \mathsf{in}\ s_1 \right]$ for some e_1, and $t_1 = e_1\,[\,[x \mapsto y]\,u\,]$. We proceed by case analysis on the other reduction. The proof of (APPLY), (TAPPLY), (OPEN) and (GET) cases are symmetric to the proof of the previous cases.

- *Case (RENAME).* Then $t = e_2 \left[\mathsf{let}_{D_2}\ x_2 = y_2\ \mathsf{in}\ s_2 \right]$ for some e_2. Now we use Lemma A.77 to analyze the relation between two focuses.
  - *When* $e_1 \left[\mathit{let}_{D_1}\ x_1 = y_1\ \mathit{in}\ s_1 \right] \preceq e_2 \left[\mathit{let}_{D_2}\ x_2 = y_2\ \mathit{in}\ s_2 \right]$. Then we have $\mathsf{let}_{D_1}\ x_1 = y_1\ \mathsf{in}\ s_1 = e \left[\mathsf{let}_{D_2}\ x_2 = y_2\ \mathsf{in}\ s_2 \right]$ for some e, and $e_2 = e_1\,[\,e\,]$. If $e = []$, then we have $t_1 = t_2$. This means that $t = e_1 \left[\mathsf{let}_{D_1}\ x_1 = y_1\ \mathsf{in}\ e'\,[\,\mathsf{let}_D\ x_2 = y_2\ \mathsf{in}\ s_2\,] \right]$ for some e'. Set t' to
  $$e_1[e'[[x_1 \mapsto y_1]\,[x_2 \mapsto [x_1 \mapsto y_1]y_2]\,s_2]].$$
  Now we inspect whether $x_1 = y_2$, and in both cases we can derive that $\gamma_1 \mid t_1 \longmapsto \gamma_1 \mid t'$, and $\gamma_2 \mid t_2 \longmapsto \gamma_1 \mid t'$, which allow us to conclude this case.
  - *When* $e_2 \left[\mathit{let}_{D_2}\ x_2 = y_2\ \mathit{in}\ s_2 \right] \preceq e_1 \left[\mathit{let}_{D_1}\ x_1 = y_1\ \mathit{in}\ s_1 \right]$. Analogous to the previous case.
  - *When* $e_1 \left[\mathit{let}_{D_1}\ x_1 = y_1\ \mathit{in}\ s_1 \right] \leftrightarrow e_2 \left[\mathit{let}_{D_2}\ x_2 = y_2\ \mathit{in}\ s_2 \right]$. Then for both directions, we can apply (RENAME) to rename the variable in the other branch, wherein the renaming in one branch is independent from the renaming in the other branch. We can therefore conclude this case.
  - *When* $e_2 \left[\mathit{let}_{D_2}\ x_2 = y_2\ \mathit{in}\ s_2 \right] \leftrightarrow e_1 \left[\mathit{let}_{D_1}\ x_1 = y_1\ \mathit{in}\ s_1 \right]$. Analogous to the previous case.
- *Case (LIFT-LET).* Then $t = e_2 \left[\mathsf{let}_{D_2}\ x_2 = v_2\ \mathsf{in}\ s_2 \right]$, $\gamma_2' = \gamma_2, \mathsf{val}_D\ x_2 \mapsto v_2$, and $t_2 = e_2\,[\,s_2\,]$. Now we apply Lemma A.77 to analyze the relation between two focuses.
  (i) *When* $e_1 \left[\mathit{let}_{D_1}\ x_1 = y_1\ \mathit{in}\ s_1 \right] \preceq e_2 \left[\mathit{let}_{D_2}\ x_2 = v_2\ \mathit{in}\ s_2 \right]$. Then, by inspecting the equality we can show that $t = e_1 \left[\mathsf{let}_{D_1}\ x_1 = y_1\ \mathsf{in}\ e \left[\mathsf{let}_{D_2}\ x_2 = v_2\ \mathsf{in}\ s_2 \right] \right]$. We set t' to
  $$e_1\,[\,e\,[\,[x_1 \mapsto y_1]\,s_2\,]\,].$$
  Importantly, since $\mathsf{cv}(v_2) \cap D_2 \subseteq \mathsf{dom}(\gamma_2)$, we can show that $[x_1 \mapsto y_1]\,v_2 = v_2$ and $[x_1 \mapsto y_1]\,D_2 = D_2$. We can derive that $\gamma_1 \mid t_1 \longmapsto \gamma_1, \mathsf{val}_{D_2}\ x_2 \mapsto v_2 \mid t'$ and $\gamma_2, \mathsf{val}_{D_2}\ x_2 \mapsto v_2 \mid t_2 \longmapsto \gamma_2, \mathsf{val}_{D_2}\ x_2 \mapsto v_2 \mid t'$. This case can thus be concluded as $\gamma_1, \mathsf{val}_{D_2}\ x_2 \mapsto v_2$ and $\gamma_2, \mathsf{val}_{D_2}\ x_2 \mapsto v_2$ are still equivalent.
  (ii) *Other cases.* In these cases, the renaming and the binding lifting do not influence each other. So in each case, for both directions we apply the corresponding rule in the other direction to conclude.
- *Case (LIFT-VAR) and (LIFT-SET).* In this two cases, we can show that the variable that is looked up is already in the store, and by the well-formedness their value does not mention the renamed variable, thus staying unaffected by the renaming. We can apply the corresponding rule to conclude each case.

Case (LIFT-LET) and (LIFT-VAR). Proceed by case analysis on the other reduction derivation. The proof of the (APPLY), (TAPPLY), (OPEN), (GET), and (RENAME) cases are again symmetric to the previous

proof. In the remaining cases, we can always swap the order the two lifted store bindings while preserving store equivalence.

Case (LIFT-SET). We do a case analysis on the other reduction derivation, wherein all but one cases can be proven symmetrically to the previous ones. The only unproven case is when both reductions are derived by the (LIFT-SET) rule. Then $t = e_1\,[\,x_1 := y_1\,]$, $\gamma_1(\mathsf{val}\ y_1) = v_1$, $\gamma_1' = \gamma_1, \mathsf{set}\ x_1 = v_1$, and $t_1 = e_1\,[\,v_1\,]$. Also, $t = e_2\,[\,x_2 := y_2\,]$, $\gamma_2(\mathsf{val}\ y_2) = v_2$, $\gamma_2' = \gamma_2, \mathsf{set}\ x_2 = v_2$, and $t_2 = e_2\,[\,v_2\,]$. By invoking Lemma A.77 we can show that either $e_1 = e_2$, or $e_1\,[\,x_1 := y_1\,] \leftrightarrow e_2\,[\,x_2 := y_2\,]$ or $e_2\,[\,x_2 := y_2\,] \leftrightarrow e_1\,[\,x_1 := y_1\,]$. In the first case we can conclude immediately since this implies that $t_1 = t_2$. Otherwise, we invoke Lemma 4.5, Lemma A.81, and Corollary A.85 to show that $x_1 \neq x_2$. Therefore, swapping the two set-bindings preserves the store equivalence. For both directions, we can apply the (LIFT-SET) rule to reduce to the same term. This case is therefore concluded. □

Definition A.7 (Reduction closures). *We define $\gamma \mid t \longmapsto^* \gamma' \mid t'$ as the reflexive and transitive closure of $\gamma \mid t \longmapsto \gamma' \mid t'$. $\gamma \mid t \longmapsto^{\leq 1} \gamma' \mid t'$ denotes the union of the reduction relation $\gamma \mid t \longmapsto \gamma' \mid t'$ and the reflexive relation. In other words, $\gamma \mid t \longmapsto^{\leq 1} \gamma' \mid t'$ means a reduction of zero or one step.*

Corollary A.87 (Diamond property of $\longmapsto^{\leq 1}$). *Given two equivalent configurations $\gamma_1 \mid t \cong \gamma_2 \mid t$, if (1) $\gamma_1 \vdash t$ and $\gamma_2 \vdash t$; (2) $\gamma_1 \mid t \longmapsto^{\leq 1} \gamma_1' \mid t_1$; and (3) $\gamma_2 \mid t \longmapsto^{\leq 1} \gamma_2' \mid t_2$, then there exists $\gamma_1'', t', \gamma_2''$ such that (1) $\gamma_1' \mid t_1 \longmapsto^{\leq 1} \gamma_1'' \mid t'$, (2) $\gamma_2' \mid t_2 \longmapsto^{\leq 1} \gamma_2'' \mid t'$, and (3) $\gamma_1'' \mid t' \cong \gamma_2'' \mid t'$.*

Fact A.88. *The transitive and reflexive closure of $\gamma \mid t \longmapsto^{\leq 1} \gamma' \mid t'$ equals $\gamma \mid t \longmapsto^* \gamma' \mid t'$.*

Lemma A.89 (Store equivalence preserves reduction). *If (i) $\gamma_1 \mid t \longmapsto \gamma_1' \mid t'$ and (ii) $\gamma_1 \cong \gamma_2$ then there exists γ_2' such that (i) $\gamma_2 \mid t \longmapsto \gamma_2' \mid t'$ and (ii) $\gamma_1' \cong \gamma_2'$.*

Proof. By straightforward case analysis on the reduction derivation, wherein in each case we apply the same typing rule. In the cases where store lookup is involved, we use Lemma A.79 and Lemma A.80 to show that the result is the same under the two equivalent stores. In the cases where the store is extended in the reduction step, we can straightforwardly show that the resulted stores are still equivalent to each other. □

Corollary A.90. *If (i) $\gamma_1 \mid t \longmapsto^* \gamma_1' \mid t'$ and (ii) $\gamma_1 \cong \gamma_2$ then there exists γ_2' such that (i) $\gamma_2 \mid t \longmapsto^* \gamma_2' \mid t'$ and (ii) $\gamma_1' \cong \gamma_2'$.*

Lemma A.91 (Asymmetric diamond property of reduction closure). *Given two equivalent configurations $\gamma_1 \mid t \cong \gamma_2 \mid t$, if (1) $\gamma_1 \vdash t$ and $\gamma_2 \vdash t$; (2) $\gamma_1 \mid t \longmapsto^{\leq 1} \gamma_1' \mid t_1$; and (3) $\gamma_2 \mid t \longmapsto^* \gamma_2' \mid t_2$, then there exists $\gamma_1'', t', \gamma_2''$ such that (1) $\gamma_1' \mid t_1 \longmapsto^* \gamma_1'' \mid t'$, (2) $\gamma_2' \mid t_2 \longmapsto^{\leq 1} \gamma_2'' \mid t'$, and (3) $\gamma_1'' \mid t' \cong \gamma_2'' \mid t'$.*

Proof. By induction on the length of the reduction $\gamma_2 \mid t \longmapsto^* \gamma_2' \mid t_2$.

When there is zero step. Then $\gamma_2' = \gamma_2$ and $t_2 = t$. We can set $t' = t_1$, then reduce t_2 one step to t_1, and reduce t_1 zero step.

When $\gamma_2 \mid t \longmapsto^* \gamma_2' \mid t_2 = \gamma_2 \mid t \longmapsto^{\leq 1} \gamma_0 \mid t_0 \longmapsto^* \gamma_2' \mid t_2$. By Corollary A.87 we can show that there exists t' such that $\gamma_1' \mid t_1 \longmapsto^{\leq 1} \gamma_1'' \mid t'$, $\gamma_0 \mid t_0 \longmapsto^{\leq 1} \gamma_0' \mid t'$, and $\gamma_1'' \cong \gamma_0'$ Now we use the preservation theorem so that we can invoke IH to show that there exists t'' such that $\gamma_0' \mid t' \longmapsto^* \gamma_0'' \mid t''$, $\gamma_2' \mid t_2 \longmapsto^{\leq 1} \gamma_2'' \mid t''$, and $\gamma_0'' \cong \gamma_2''$. Now we invoke Corollary A.90 to show that there exists γ_1''' such that $\gamma_1'' \mid t' \longmapsto^* \gamma_1''' \mid t''$, and $\gamma_1''' \cong \gamma_0''$. Therefore, we can show that $\gamma_1' \mid t_1 \longmapsto^* \gamma_1''' \mid t''$. Also, we can show that $\gamma_1''' \cong \gamma_2''$, and thus conclude this case. □

Proof. By induction on the length of the first reduction.

When the length is zero. This case can be trivially concluded.

When $\gamma_1 \mid t \longmapsto^{\leq 1} \gamma_0 \mid t_0 \longmapsto^* \gamma_1' \mid t_1$. We first invoke Lemma A.91 to show that $\exists t'$ such that $\gamma_0 \mid t_0 \longmapsto^* \gamma_0' \mid t'$, $\gamma_2' \mid t_2 \longmapsto^{\leq 1} \gamma_2'' \mid t'$, and $\gamma_0' \cong \gamma_2''$. Then, we use the preservation lemma so that we can invoke IH, showing that $\exists t''$ such that $\gamma_1' \mid t_1 \longmapsto^* \gamma_1'' \mid t''$, $\gamma_0' \mid t_0 \longmapsto^* \gamma_0'' \mid t''$, and $\gamma_1'' \cong \gamma_0''$. Now by Corollary A.90 we can show that $\exists \gamma_2'''$ such that $\gamma_2'' \mid t' \longmapsto^* \gamma_2''' \mid t''$ and $\gamma_2''' \cong \gamma_0''$. Therefore, we have $\gamma_2' \mid t_2 \longmapsto^* \gamma_2''' \mid t''$ and can show that $\gamma_2''' \cong \gamma_1''$, thus concluding this case. □

Lemma A.92 (Answers do not reduce). *Given any store* γ *and an answer* a, $\gamma \mid a \longmapsto \gamma' \mid t'$ *is impossible.*

Proof. By straightforward case analysis on the reduction derivation, wherein none of the rules reduces an answer. □

Corollary A.93. *Given any store* γ *and an answer* a, $\gamma \mid a \longmapsto^* \gamma' \mid t$ *implies that* $\gamma' = \gamma$ *and* $t = a$.

Theorem 4.7 (Uniqueness of Answer). *For any* t, *if (i)* $\gamma \vdash t$, *(ii)* $\gamma \mid t \longmapsto^* \gamma_1 \mid a_1$ *and (iii)* $\gamma \mid t \longmapsto^* \gamma_2 \mid a_2$ *then* $a_1 = a_2$ *and* $\gamma_1 \cong \gamma_2$.

Proof. We first use Theorem 4.6 to show that $\exists \gamma_1', \gamma_2', t'$ such that $\gamma_1 \mid a_1 \longmapsto^* \gamma_1' \mid t'$ and $\gamma_2 \mid a_1 \longmapsto^* \gamma_2' \mid t'$ where $\gamma_1' \cong \gamma_2'$. Now we use Corollary A.93 to show that $t' = a_1 = a_2$, $\gamma_1 = \gamma_1'$ and $\gamma_2 = \gamma_2'$, which conclude our goal. □

Received 21-OCT-2023; accepted 2024-02-24